APL

With a Mathematical Accent

Clifford A. Reiter

William R. Jones

Quick-Reference Function Table

Symbol	Monadic Dyadic	Page	Name
Chapter One			
+	D	1	Plus
×	D	1	Times
÷	D	1	Divide
*	D	1	Power
−	D	2	Minus
=	D	4	Equals
−	M	5	Negate
÷	M	5	Reciprocal
×	M	5	Signum
*	M	5	Exponential
⊛	M	5	Natural logarithm
⊛	D	5	Logarithm
|	M	5	Absolute value
⍴	M	7	Shape
f/	M	7	Reduction (operator)
⍳	M	8	Index generator

Chapter Two

⍴	D	14	Reshape
f/	M	14	Reduction (operator)
∘.*f*	D	15	Outer product (operator)
,	D	16	Catenate
[ ]	D	18	Index
⍴	M	20	Shape

Chapter Three

?	M	30	Roll

Chapter Four

+.×	D	41	Matrix product
⌹	D	44	System solver
∘.*f*	D	45	Outer product (operator)
⌹	D	47	Least-squares system solver
⌹	M	49	Matrix inverse
⍉	M	49	Transpose

Chapter Five

<	D	57	Less than
≤	D	57	Less than or equal
≥	D	57	Greater than or equal
>	D	57	Greater than
≠	D	57	Not equal
~	M	59	Not
∧	D	59	And
∨	D	59	Or
⍲	D	59	Nand
⍱	D	59	Nor
f.*g*	D	60	Inner product (operator)
⌊	M	64	Floor
⌈	M	64	Ceiling
∈	D	65	Membership

Chapter Six

?	M	69	Roll
?	D	70	Deal
!	M	74	Factorial
!	D	74	Binomial
○	M	78	Pi times
○	D	77	Circle (includes trig functions)

Chapter Seven

⌈	D	82	Maximum
⌊	D	82	Minimum
⍋	M	83	Grade up
/	D	86	Replicate
f\	M	88	Scan (operator)

Chapter Eight

↑	D	99	Take
↓	D	99	Drop
,	M	101	Ravel
/	D	102	Replicate
\	D	103	Expansion
⌽	M	104	Reverse
⌽	D	104	Rotate
⍉	M	106	Transpose
⍉	D	106	Transpose
,[ ]	D	108	Laminate

Chapter Nine

⍋	M	114	Grade up
⍒	M	114	Grade down
⍳	D	115	Index relative to
|	D	119	Residue
⍒	D	121	Represent
⊥	D	122	Base value

↓	M	149	Format
↓	D	150	Format
↑	M	152	Execute

APL

With a Mathematical Accent

Clifford A. Reiter
William R. Jones
Lafayette College

Brooks/Cole Publishing Company
Pacific Grove, California

Brooks/Cole Publishing Company
A Division of Wadsworth, Inc.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data

Reiter, Clifford A., [date]
APL with a mathematical accent / Clifford A. Reiter and William R. Jones.
p. cm.
Includes bibliographical references.
ISBN 0-534-12864-5 [price]
1. APL (Computer program language) I. Jones, William R., [date]
. II. Title.
QA76.73.A27R45 1990 89-22344
510'.28'55133 — dc20 CIP

Sponsoring Editor: *John Kimmel*
Marketing Representative: *Winston Beauchamp*
Editorial Assistant: *Mary Ann Zuzow*
Production Editor: *Ben Greensfelder*
Manuscript Editor: *Harriet Serenkin*
Permissions Editor: *Carline Haga*
Interior and Cover Design: *Michael Rogondino*
Art Coordinator: *Lisa Torri*
Interior Illustration: *Lori Heckelman*
Typesetting: *Beacon Graphics*
Printing and Binding: *Malloy Lithographing, Inc.*

To Marilyn
—Cliff

To Betsy May
—Bill

Preface

APL with a Mathematical Accent is an introduction to the computer language APL (A Programming Language), providing quick access to the powerful computational capabilities of the language for the newcomer to APL. The book's organization capitalizes on the mathematical nature of the language, and complements standard curricula. A broad sampling of mathematical applications is included.

The book can be used as a main text for a course that uses computers for solving mathematical problems or doing mathematical experimentation. Alternatively, portions of the book can serve as an adjunct text to provide methods for expeditious handling of computing problems.

APL with a Mathematical Accent lets students quickly acquire computational skills applicable to various subject areas, particularly linear algebra, statistics, probability, numerical analysis, operations research, abstract algebra, number theory, and mathematical modeling. Experience has shown, for example, that using this book as an adjunct text can enhance a linear algebra course at a cost in time of one or two lectures early in the term, followed by about fifteen minutes spent on any particular APL functions needed for a given computer assignment. Selected reading assignments from chapters 1 through 4 give sufficient preparation for computer solution of substantial problems from linear algebra. Flexibility exists for solving these problems using only primitive APL functions or using user-defined functions.

The organization of *APL with a Mathematical Accent* facilitates looking up unfamiliar items and reviewing topics by

- Providing a quick-reference function table
- Keeping sections that introduce topics short and independent
- Providing a contents and index
- Including error message and debugging notes in an appendix
- Providing system functions and variables in an appendix

Individual functions, operators, et cetera, are first introduced and illustrated simply and independently of each other. A few topics are introduced in more than one place to preserve independence of sections. Then significant examples, often involving a topic from college-level mathematics, are given. These examples provide solutions for basic types of computer problems and illustrate good APL usage.

The end-of-chapter exercises range from simple reviews of fact to challenging problems. Answers to underlined exercises are provided. Answers to all exercises are available by writing to Brooks/Cole.

Overview

Chapter 1 introduces the syntax of APL. It presents many of APL's primitive mathematical functions and applies them to vector arguments. The chapter also introduces the reduction operator and shape and index generator, allowing the presentation of examples of useful computations that begin to give the flavor of APL computing. The chapter closes with a brief overview of the workspace environment and system commands.

Chapter 2 treats numeric arrays and manipulations. It includes elementwise arithmetic, reshape, matrix reductions, outer products, catenation, and indexing and uses the manipulation of arrays as entities. Chapter Two also discusses the shape function and introduces empty arrays.

Chapter 3 introduces user-defined functions that have explicit arguments and return a result usable by other functions. This "mathematical type" of user-defined function is used until Chapter 10, where "programming type" functions are developed. Also discussed in Chapter 3 are editing of functions, frequently used system commands, and functions with simple loops. Some features of user-defined functions (e.g., line labels and comments) that are in Chapter 10 may be studied along with this chapter.

APL functions especially useful in matrix algebra are presented in Chapter 4. The chapter treats matrix product, matrix inverse, system solver, and transpose. It reviews and uses outer product and least-square solutions of linear systems.

Chapter 5 treats the numerical relational functions (comparatives) and the logical functions. It develops generalized inner products with many examples on vectors and matrices and illustrates numerous tabulating capabilities of these functions. The floor, ceiling, and membership functions are introduced, and the system variable comparison tolerance is discussed.

Simulation using the "randomizing" functions is a principal theme of Chapter 6. The roll, deal, factorial, and binomial functions are treated, and random link is discussed. The circular, hyperbolic, and pythagorean functions are presented. Many examples in this chapter use the relational functions from Chapter 5.

The principal topics of Chapter 7 are descriptive statistics and character-based computer graphing. The chapter includes maximum, minimum, grade up, and replicate functions as well as the scan operator. Character arrays are introduced for use in graphing, and applications to data analysis and sampling from populations are included.

Array manipulation and construction is the theme of Chapter 8. The functions take, drop, ravel, replicate, expansion, compression, reverse, rotate, and laminate are presented. Applications to polynomial manipulations are illustrated. Higher dimensional arrays receive more treatment here than in previous chapters.

The themes of Chapter 9 are orderings and representations. The sorting functions grade up, grade down, and dyadic iota are treated. The residue, represent, and base value functions are presented, and index origin and atomic vector are discussed. A working version of an error-correcting Hamming code completes the chapter.

Chapter 10 provides tools for writing complex functions and programs. User-defined functions of the programming type are emphasized. Various modes of branching, line labels, comments, quad, and quote-quad are discussed and amply illustrated. The format and execute functions are treated. Complex sample programs are given.

Appendix A is an introduction to APL's effective debugging features. Appendix B shows how to use and manage workspaces. It also has a table of system variables and functions and brief descriptions of their uses.

The bibliography has a wide range of sources on APL. In particular, it has books that are suitable as follow-ups to this one for different areas of APL, including those by Berquist; Brown, Pakin and

Polivka; and Gilman and Rose. It also lists books that apply APL to specific subjects as well as journals that provide current information about APL, including vendors and products.

Acknowledgments

We extend sincere thanks to Linda Alvord, Kenneth Iverson, James Lipscomb, and Charles Sims for their roles in making APL a part of our lives. Many thanks are also expressed for the helpful comments from prepublication reviewers Roger Glassey, of the University of California at Berkeley; Gary Helzer, of University of Maryland, College Park; and Peter Lewis of the U.S. Naval Postgraduate School, Monterey, California. Special thanks go to Claudette Dahlinger who did large portions of the demanding typing job. We are grateful for the contributions that student reactions and discussions have had on the concept of the book. Finally, our thanks to Harriet Serenkin for her copy editing work, and to the staff at Brooks/Cole — most notably John Kimmel and Ben Greensfelder — for their efforts in the publication of this book.

Clifford A. Reiter
William R. Jones

Contents

APL

With a Mathematical Accent

1

First Steps With APL

Chapter One begins by discussing how to interact with APL and interpret basic APL statements. It gives several familiar mathematical functions and their APL denotations. It also introduces some of APL's powerful vector processing capabilities and special functions. Finally, the chapter briefly discusses some facts about the APL work environment and its management.

1.1 Interacting With APL

Using APL can be somewhat like using a calculator. You enter data or commands, and APL responds. If you enter `3+2`, for example, the response is `5`. The ongoing exchange between you and APL is recorded on paper or on a video screen. The record, or *session log,* of a few arithmetic problems could look like this:

```
      3×2        You type 3×2, then press the Return key.
6                APL gives this response.
      3÷2        You enter 3÷2.
1.5              APL responds.
      3*2        You enter 3*2.
9                APL responds.
```

The APL response begins at the left margin on the line below your entry. After the response there is an automatic advance to the next line and a six-space indent; then APL is ready for further input. This format makes it easy to distinguish your input from APL's response. This book has several small

examples from a session log that are printed across the page like this:

```
      3×4                   3÷4                   3*4
12                    0.75                  81
```

1.2 Order of Execution

Common mathematical notation has rules governing the order in which mathematical operations are performed. For example, multiplication and division take precedence over addition and subtraction. The commonly used rules interpret $3 \times 4 - 5$ as $12 - 5$, not $3 \times (-1)$ and $2 + 6 \div 3$ as $2 + 2$, not $8 \div 3$. Conventions for interpreting more complicated expressions such as $\sin^2 ab$ also exist, but they are intricate to codify and can lead to ambiguous expressions when many functions are involved.

APL has but one rule governing the order of function execution within APL expressions. There is no hierarchy of functions with some taking precedence. The resulting simplicity is especially important because APL has many *primitive* (built-in) functions. Simply expressed the rule is

APL functions are executed in order from right to left.

Note that the rule is that you start at the right and work to the left. For example, the APL interpretation of the expression `3×4+5` is as follows: First add the `4` and `5` to get `9` (the rightmost function is +), then multiply `9` by `3` to get `27`. Thus, the APL evaluation of `3×4+5` is `27`. As with conventional mathematical notation, expressions may be grouped with parentheses; the right-to-left rule also applies within the parentheses.

Here are some examples; note the effect of parentheses:

```
      3×4+5                 3×(4+5)               (3×4)+5
27                    27                    17

      8÷2+2                 (8÷2)+2               12÷4-3
2                     6                     12
```

The order of execution rule applies to all functions; here are examples with the power function or exponentiation, denoted *:

```
      2*3+2                 (2*3)+2               2*3*2
32                    10                    512
```

Roots are treated as fractional powers in APL; for example,

```
      9*.5                  9*1÷2                 2*.5
3                     3                     1.414213562
```

1.3 Negative Sign, Minus Sign

Negative numbers are indicated with a high minus sign; for example, `¯2`. Note the difference in position between the minus sign that indicates the arithmetic function subtraction, as in `3-5`, and the

APL negative sign that serves as part of the symbol denoting a negative number, as in ¯2. Here are examples:

```
      4-7                  2-4+6                2-3*2
¯3                   ¯8                   ¯7

      ¯4+7                 ¯3×¯4                ¯3*3
3                    12                   ¯27

      ¯3*2                 ¯3*¯2                2*¯2
9                    0.1111111111         0.25
```

1.4 Assignment and Variable Names

APL variables are given values by *assignment,* which is symbolized with a left-pointing arrow as in A←5. When you enter A←5, 5 is stored with the name A, the automatic advancement and indentation occurs in the session log, and APL awaits your next entry. After a variable has been assigned a value, entering that variable name alone results in the display in the session log of the current value of the variable. A new assignment to a variable name replaces any value formerly stored with that name. Here is a sample session log:

```
      A←5
      B←1+2*3
      B                Display of B is requested.
9
      A+B
14
      A←A+B            A is given a new value.
      A
14
      2×C←A-B          C is assigned a value within an expression.
10
      C
5
```

APL displays the result of the last function applied in a line unless the last operation is assignment. Multiple assignments are convenient; for example, K←J←1 would assign both K and J the value 1. The expression F←2+E←D×D←3 would assign the values 3, 9, and 11 to the variables D, E, and F, respectively.

More suggestive variable names can be helpful; for example, to find the area of a rectangle,

```
      BASE←10.7
      HEIGHT←8.1
      AREA←BASE×HEIGHT
      AREA
86.67
```

A letter followed by any combination of letters and numerals is a valid variable name; typically, systems allow up to 77 characters in a name. Thus, AA, B14C, X5Y05, Z3TERM, and MOON are

valid variable names. Modern systems usually have uppercase and lowercase alphabets, permitting ample flexibility in constructing names.

1.5 Example: Heron's Formula

Suppose A, B, and C are the lengths of the sides of a triangle and S is its semiperimeter. The area of the triangle is given by Heron's formula, which in common notation is

$$\text{Area} = \sqrt{S(S-A)(S-B)(S-C)}, \qquad S = \frac{1}{2}(A+B+C)$$

The corresponding formulas in APL are

```
S←.5×A+B+C
AREA←(S×(S-A)×(S-B)×S-C)*.5
```

1.6 The Equals Function

The APL *equals* function tests whether expressions have the same value and gives the result 1 to indicate true or 0 to indicate false. Equals is not used for assignment. With `A←3` and `B←7` we have

```
      3=A                A=2                A=B-4
1                  0                  1

      B=A+4              A+4=B              B-4=A
1                  3                  7
```

(Remember to execute right to left.)

1.7 Examples: Right Triangle Test, Conditional Incrementing

(a) If `X`, `Y`, and `Z` are the side lengths of a triangle, with `Z` largest, you could test whether it is a right triangle with the expression `(Z*2)=(X*2)+Y*2`.

(b) If you wished to increment a variable `N` by 3 if variable `X` equals `N` but leave `N` unchanged otherwise, the expression `N←N+3×X=N` would do that. This type of expression is useful in function definition (programming).

1.8 Dyadic, Monadic Functions

Each of the functions introduced thus far has its symbol appear between two numbers or variable names, called *arguments* of the function. For example, in the expression `3+5`, 3 and 5 are arguments of the addition function; in `2=A`, 2 and `A` are arguments of the equals function. APL functions that require two arguments are called *dyadic* functions. There are many dyadic functions, but there are many other APL functions that have only one argument; these are called *monadic* functions. ***The argument of a monadic function is placed on the right of the function symbol.*** Two basic mathematical

functions that are monadic APL functions are negation and reciprocal. Negation is symbolized by the same minus sign as used for subtraction and yields the negative of its argument. Reciprocal is symbolized by the ordinary division sign, ÷, and yields the reciprocal of its argument. For example,

```
      -6          -¯6          --6
¯6            6            6

      ÷2          ÷÷2          ÷-2
0.5           2            ¯0.5
```

Because there are many primitive functions in APL, many of the function symbols serve "double duty" and signify a monadic or dyadic function according to context. The monadic interpretation of a function symbol applies if there is another function symbol immediately to the left of it or if it is the leftmost symbol within an expression; otherwise, the dyadic interpretation is understood. (Later the introduction of operators will modify this dichotomy.) Thus, in `6÷-2` and `-3÷2` the monadic minus or negation is signified, but in `6-÷2` the dyadic minus or subtraction is understood; for example,

```
      6÷-2          -3÷2          6-÷2
¯3            ¯1.5          5.5
```

Some other monadic functions are introduced next. Monadic × denotes the signum (or sign) function of mathematics that gives the value `¯1` for negative arguments, `+1` for positive arguments, and `0` for `0` argument; for example,

```
      ×2-5          ×2=3          ×2+4
¯1            0             1
```

The monadic use of * signifies the exponential function, e^x in common notation. Thus, `*0` yields `1` and `*1` yields `2.718281828`; if *X* has an assigned value *x*, then `*X` yields e^x. The inverse of the exponential function is the natural logarithm function, ln *x*, which in APL is denoted by the monadic use of ⍟ (circle star). Thus, `⍟1` is `0` and for any real number *X* the expression `⍟*X` yields *X*. The dyadic use of ⍟ signifies the logarithm to a base specified by the left argument. Thus, `2⍟8` is `3` and `8⍟2` is `0.3333333333`.

Absolute value or magnitude is a basic mathematical function, which in APL is symbolized by the monadic use of the stile | or vertical bar as follows:

```
      |0          |3-5          |5-3
0             2             2
```

For example, on a coordinate line, the distance between points with coordinates `X1` and `X2` is given by `|X1-X2`.

APL has special symbols for many functions; a quick reference table for these symbols is inside the cover.

1.9 *Functions With Vector Arguments*

A numeric *vector* is a list or ordered set of numbers. If `V←2  0.3  ¯2` (note the spaces), then *V* is a vector with three elements or components—namely, the *scalars* `2`, `0.3`, and `¯2`. A great strength of APL is that many functions that take scalars as arguments can also take vectors as arguments; the function is applied elementwise and the result is a vector. Such functions are called *scalar functions*.

Examples are

```
      A←0 2 4
      A
0 2 4

      B←5 ¯2 1
      B
5 ¯2 1

      A+B
5 0 5

      A×B
0 ¯4 4

      A÷B
0 ¯1 4

      A*B
0 0.25 4

      -B
¯5 2 ¯1

      ÷B
0.2 ¯0.5 1

      |B
5 2 1
```

If one argument of a dyadic function is a scalar and one is a vector, the scalar is treated as a vector of matching length to the given vector and with all elements equal. For example,

```
      A
0 2 4

      B
5 ¯2 1

      2+5 ¯1 2
7 1 4

      5+A
5 7 9

      5 5 5+A
5 7 9

      3×B
15 ¯6 3

      2*A
1 4 16

      A*3
0 8 64

      2=A
0 1 0
```

1.10 *Examples: Some Uses of Vector Arithmetic*

Computations arising in many and varied settings can be efficiently handled with vector arithmetic.

(a) If *A*, *B*, and *C* are position vectors for the vertices of a triangle in space of any dimension, then `(A+B+C)÷3` gives the position vector of the centroid of the triangle. For example,

```
      A
6 0 0

      B
0 2 1

      C
0 1 2

      (A+B+C)÷3
2 1 1
```

Compute centroid of triangle *ABC*.

(b) If *TIME* is a vector each component of which is the time in hours for each of several cars to complete a 350-mile course, then `SPEED←350÷TIME` is a vector of the speeds (in miles per hour) over the course for the cars. Here is an example:

```
      TIME←3.825 3.887 3.606 3.791 3.711
      SPEED←350÷TIME
      SPEED
91.503 90.044 97.060 92.324 94.314
```

(c) In an investigation of the impact of a fish species on lake ecologies, a biologist had need to calculate the areas of large numbers of (roughly) elliptical fish nests. (Recall that the area of an ellipse is $\pi \times A \times B$, where A and B are the semimajor and semiminor axes of the ellipse.) If the semiaxes measurements of the ellipses are stored as corresponding entries of two vectors labeled

SMAJOR and *SMINOR*, then the expression

```
AREA←3.14×SMAJOR×SMINOR
```

computes and stores the approximate areas of the nests.

1.11 Shape, Reduction Operator

Two operations often applied to lists of numbers are counting the elements in the list and adding the elements in the list. These and many other operations on lists are made easy with special functions and operators, two of which are introduced here. The *shape* function, denoted by monadic use of rho, ρ, when applied to a vector gives the *length* (number of elements) of the vector. Thus, with `A←¯2 ¯1 0 4 7` shape yields

```
      ρA                ρ5 1 4              ρ0 0
5                   3                   2
```

More precisely, if *V* is any vector, then ρ *V* is a single element vector, the sole entry being the length of *V*.

The *plus reduction* of a vector, symbolized +/, gives the sum of the elements in the vector. The result of `+/1 ¯2 4 4` is exactly the same as the result of `1+¯2+4+4` — namely, 7. Examples are

```
      +/2 5 9             A                   +/A
16                  1 ¯2 4 4            7

      +/A+2             A=4                 +/A=4
15                  0 0 1 1             2
```

Notice the expression +/*A*=4 counts the number of 4s in *A*.

Two basic list processing computations are illustrated next; these are very simple to perform using APL.

1.12 Examples: Average, Dot Product

(a) Since +/*V* is the sum of the elements in *V* and ρ *V* is the number of elements in *V*, then (+/*V*)÷ρ*V* gives the average of the elements in *V*. Thus, if `V←1 ¯2 6 3`, then (+/*V*)÷ρ*V* yields 2. Applying this formula for average to the fish nest problem [Section 1.10, Example (c)], gives (+/*AREA*)÷ρ*AREA* as the average area of the fish nests.

(b) The dot (or scalar) product of two vectors of matching length is useful in various applications. If $U = (u_1, u_2, \ldots, u_n)$ and $V = (v_1, v_2, \ldots, v_n)$ are vectors, then $U \cdot V = u_1v_1 + u_2v_2 + \cdots + u_nv_n$ gives the dot product of U and V. In APL, the dot product is simply +/*U*×*V*. For example,

```
      U←1 2 0 4 3
      V←4 3 1 1 ¯1        Products of corresponding entries.
      U×V                 Add the terms of U×V.
4 6 0 4 ¯3
      +/U×V
11
```

1.13 More on the Reduction Operator

Any dyadic scalar function may be used with the *reduction operator*, /. *Minus reduction*, -/, is especially useful because it yields what in mathematics is called an alternating sum. For example, -/1 2 3 4 in APL means 1-2-3-4, which is equivalent in APL to 1-(2-(3-4)), which is equivalent ***in common notation*** to 1 − 2 + 3 − 4. Notice the alternating signs.

If *f* is any dyadic scalar function and *V* is any vector with two or more elements, then *f*/*V* produces the same result as putting an *f* between successive pairs of elements in *V*. The result is a scalar; the vector is "reduced" to a scalar. Thus, ×/2 2 3 is the same as 2×2×3, which yields 12 and

```
      -/2 2 3          */2 2 3          ÷/2 2 3
3                256                3
```

Notice that the reduction operator is different from any APL function in that a function has scalars or vectors (and, later, other arrays) as arguments, whereas the reduction operator has a function as its argument! APL operators are characterized by their use of functions as arguments. Operators generate new functions or modify the ways functions are used.

1.14 Index Generator

Generating a list of numbers with a prescribed pattern is involved in a great variety of problems. For example, to calculate the values of a function at every .2 of a unit on the interval from 1 to 4, the generation of the numbers 1, 1.2, 1.4, . . . , 3.8, 4 is part of the problem. A structured list of numbers is used in representing the sum of the first 100 terms of the alternating series as follows

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + - \cdots - \frac{1}{199} + \cdots$$

Such problems are readily handled with the *index generator* function, which is denoted by the monadic use of iota, ⍳. If *N* is any positive integer ⍳*N* is the vector of all integers from 1 to *N* in natural order:

```
      ⍳3              ⍳6              ¯1+⍳6
1 2 3           1 2 3 4 5 6     0 1 2 3 4 5

      2*⍳6            (⍳6)*2          3+.1×⍳5
2 4 8 16 32 64  1 4 9 16 25 36  3.1 3.2 3.3 3.4 3.5
```

Of course, the index generator function can be used to generate long vectors. These long vectors are "wrapped" onto several lines for display; for example,

```
      ⍳75
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
      25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
      44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62
      63 64 65 66 67 68 69 70 71 72 73 74 75
```

1.15 Examples: A Partial Sum, a Sequence for e

The index generator function can be used to calculate the alternating sum of the reciprocals of the first 100 odd integers mentioned above. Notice that `¯1+2×⍳100` gives the vector of the first 100 odd integers. Next take reciprocals with monadic ÷, and finally get the alternating sum with `-/`, giving the formula `-/÷¯1+2×⍳100` for the desired sum.

Consider the following sequence that converges to *e*, the base of the natural logarithm system:

$$\left\{\left(1+\frac{1}{1}\right)^{1},\left(1+\frac{1}{2}\right)^{2},\left(1+\frac{1}{3}\right)^{3},\ldots\right\}$$

An expression for the first *N* terms of that sequence is `(1+÷⍳N)*⍳N`; for example,

```
      N←10
      (1+÷⍳N)*⍳N
2 2.25 2.3704 2.4414 2.4883 2.5216 2.5465 2.5658 2.5812 2.5937
```

1.16 Number Forms and Print Precision

Your numerical entries and APL numerical responses are in decimal notation. For integers, the decimal point is omitted. Scaled form (exponential notation) may be used. For example, instead of writing `.00000000205`, you may write `2.05E¯9`. In some cases, APL output will be in scaled form. Of course, you can control the form of the output; see Section 10.15 and Appendix B.5 for further details. One choice of output control is the number of significant digits to be printed, and one of the ways to make that choice is through the value of the system variable *print precision*, `⎕PP` (pronounced "quad-P-P"). Print precision has a default value already assigned when you sign on to APL, but you may change it to any positive integer within the limits of your APL system, typically from 1 to 17. An example session using `⎕PP` is as follows:

```
      A←5÷3
      A
1.666666667
      ⎕PP                Display the current value of ⎕PP.
10
      ⎕PP←3              Reassign ⎕PP.
      A
1.67
      ⎕PP←15             The number of significant figures displayed
      A                  changes; the value of A does not change.
1.66666666666667
```

See Appendix B.4 for a list of system variables.

1.17 Workspaces, System Commands

When you sign on an APL system, a block of computer memory is set aside for your use; it is called your *active workspace*. Any variables you assign or functions (programs) you define will be stored in

this workspace. Such variables and functions remain available to you until you redefine or erase them, replace your active workspace, or end your work session without saving them.

System commands are used for managing your workspaces. For example, if you wish to review the variable names you have in your active workspace, you enter the system command)*VARS* and a list of those names is printed on the session log. Similarly,)*FNS* causes a list of the names of the user-defined functions in your active workspace to be printed. All system commands have a right parenthesis as the initial character.

Other system commands allow you to easily save your entire active workspace for use in subsequent work sessions, retrieve previously saved workspaces, copy work from saved workspaces into your active workspace, and perform other management functions on your APL workspaces. Frequently used system commands are discussed in Section 3.8; all are treated in Appendix B. One more will be mentioned here.

1.18 Exiting APL

When you have completed a work session and ***after*** you have saved anything you might want for later reference, you terminate your work session with the system command)*OFF*. Upon entering)*OFF*, the APL system may print some summary information about the session, and you will be properly disconnected from the APL system.

Exercises

1. Determine the APL evaluation of each expression; check your answers with a computer.

(a) 3×4−4	(b) (3×4)−4	(c) 2*1*3	(d) (2*1)*3
(e) 2+2=3	(f) 3=2+2	(g) 2*0	(h) 0*2
(i) A+B←3×A←2	(j) 2×N+4=N←5	(k) −0	(l) ¯0
(m) −¯2	(n) ×2	(o) ÷10	(p) *1
(q) ⊛1	(r) *⊛5	(s) ∣¯5+2	(u) −×2

2. Determine the APL evaluation of each expression; check your answers with a computer.

(a) 3÷−1	(b) 3−÷1	(c) 3×−2	(d) 3−×2
(e) 3×÷3	(f) 3÷×3	(g) −3×−2	(h) ×3×−2
(i) −3×−¯3	(j) −3−×¯3	(k) 3×÷3÷×3	(l) ÷3÷3÷÷3

3. Indicate whether each use of a function in Exercise 2 is monadic or dyadic.
4. Determine the APL evaluation of each expression; check your answers with a computer.

(a) 2 2+1	(b) 3+2 2+1	(c) ¯4+⍳7	(d) 4−⍳7
(e) (⍳5)÷5	(f) ⍳5÷5	(g) ¯10+10×⍳6	(h) 0×⍳6
(i) ⍳1	(j) −1 2 4	(k) −/1 2 4	(l) ×1 2 4
(m) ×/1 2 4	(n) ÷1 2 4	(o) ÷/1 2 4	(p) */4 1 2

5. Let A←1 3 2 3 1 0 0 3 3 8 3. Evaluate: (a) ⍴A (b) ⍴⍴A (c) +/A (d) A=3 (e) +/A=3 (f) +/A=0 (g) +/A=5.
6. Let B←⍳13 and U←1 0 1 0 1 0 1 0 1 0 1 0 1. Evaluate B×U and B*U; describe in words the results of +/B×U and ×/B*U.

7. Observe the error messages produced in response to these entries:

   (a) `1÷0` (b) `÷0` (c) `2 2÷1 0` (d) `2(3+4)`
   (e) `(2+3)4` (f) `1 2+3 4 5` (g) `0 0 0×0 0` (h) `¯A←2`
   (i) `¯×2` (j) `⍳¯2` (k) `⍳1 4` (l) `1 2 3+⍳4`

8. In mathematics $0 \div 0$ is undefined and 0^0 has a value only in some contexts. Try these in APL: `0÷0` and `0*0`.
9. Write brief APL expressions that yield these vectors:

   (a) 1 1.2 1.4 ··· 3.6 3.8 4
   (b) 1 .5 0 ¯.5 ··· ¯10.5 ¯11

10. Write brief APL expressions that yield the sum of the first 100 terms of each series: (a) $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots$ (b) $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + - \cdots$ (c) $1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots$
11. Let V be any vector. Write brief APL expressions for the:

   (a) sum of the squares of the elements of V
   (b) sum of the absolute values of the elements of V
   (c) alternating sum of the elements of V
   (d) number of zero elements of V
   (e) number of positive elements of V (Hint: Use signum function)
   (f) sum of the positive elements of V

12. Incorporate the dot product formula `+/U×V` of Example 1.12 in an APL formula for the vector projection of U onto V. Find the vector projection of `U←1 0 8` onto `V←2 ¯2 2` using the formula.
13. Let a system of point masses $m_1, \ldots, m_k$ be located in the plane at $(x_1, y_1), \ldots, (x_k, y_k)$. The total moment of the system about the y axis is $\sum m_i x_i$ and about the x axis is $\sum m_i y_i$, where sums are taken from 1 to k. The x coordinate of the center of mass is $(\sum m_i x_i) \div \sum m_i$ and the y coordinate is $(\sum m_i y_i) \div \sum m_i$. With M as the vector of masses, X as the vector of x coordinates, and Y as the vector of y coordinates, write APL formulas for the total moments and the coordinates of the center of mass. Use the formulas to find the center of mass of the system of point masses: $m_1 = 1.55$, $m_2 = 0.75$, $m_3 = 0.4$, $m_4 = 2.15$ located at the points $(0, 0)$, $(3.8, 0)$, $(10, 2.2)$, $(-2, 3.5)$.

2

Fundamentals of Arrays

APL does a wonderful job of handling arrays. In this chapter all the arrays will be numeric; see Chapter 7 for character arrays. The simplest arrays are scalars, vectors, and matrices. A scalar is a number, a vector is a list of numbers, and a matrix is a rectangular arrangement of numbers. Higher dimensional arrays are also available and will be considered in the exercises and in later chapters. The functions on vectors introduced in Chapter 1 are extended to matrices, and basic types of array construction and manipulation are considered. These tools are quite powerful. The first example, which uses the vector ideas from Chapter 1, gives a glimpse of how valuable an "array view" of a computation can be when working with APL.

2.1 Example: Polynomial Evaluation

Suppose the polynomial $P(t) = 3 + 2t - 5t^2 + t^4$ is given and you are interested in evaluating P for any t value. Several APL expressions to do this are considered; in them, let `T←10`. Thus, a direct APL expression for $P(10)$ is

```
      3+(2×T)+(¯5×T*2)+T*4
9523
```

Horner's method for evaluating a polynomial requires fewer arithmetic operations; it is effected by factoring a t out of successive terms, and in common notation is $3 + t(2 + t(-5 + t(t)))$. In APL this can be expressed without parentheses:

```
      3+T×2+T×¯5+T×T
9523
```

A very different approach is to use vector arrays. Polynomial evaluation can be viewed as the sum of the elements of the vector that consists of the monomial terms. The vector of monomials is

the elementwise product of the vector of coefficients of the polynomial with the vector of powers of T.

```
      T*0 1 2 4                    These are the powers of T.
1  10  100  10000
      3 2 ¯5 1×T*0 1 2 4           Multiply by the coefficients.
3  20  ¯500  10000                  This is the vector of monomial terms.
      +/3 2 ¯5 1×T*0 1 2 4         Add the terms.
9523
```

Putting in a zero coefficient for the t^3 term gives

```
      +/3 2 ¯5 0 1×T*0 1 2 3 4
9523
```

The index generator function will shorten this expression in a way that will also help to generalize it. Recall that `¯1+⍳5` is `0 1 2 3 4`. Therefore, $P(10)$ can be calculated by

```
      +/3 2 ¯5 0 1×T*¯1+⍳5
9523
```

Now if `T←¯3`, then $P(-3)$ can be evaluated by the same expression:

```
      +/3 2 ¯5 0 1×T*¯1+⍳5
33
```

In Chapter 3 this method will be incorporated into a function for polynomial evaluation. In Section 2.6 an expression that allows a polynomial to be evaluated at several T values at once is considered.

2.2 *Elementwise Arithmetic*

Consider the matrices A and B; inputting matrices is treated below. Scalar functions apply elementwise to matrices as they did to vectors; for example,

```
      A              B              A+B
1 2 3          1  2  0        2 4 3
4 5 6          2 ¯1  0        6 4 6

      A*B            A×B            A=B
 1   4    1        1  4  0        1 1 0
16  0.2   1        8 ¯5  0        0 0 0

      -B             ×B             ×A
¯1 ¯2  0          1  1  0        1 1 1
¯2  1  0          1 ¯1  0        1 1 1
```

Notice that $A \times B$ gives the elementwise product of A and B but not the matrix product, which is considered in Chapter 4. If one of the arguments is a scalar, then it is paired with each element of the other argument; for example,

```
      3×B            A*2            60÷A
3  6  0            1  4  9        60 30 20
6 ¯3  0           16 25 36        15 12 10
```

2.3 *Reshape: Defining Arrays*

Matrices can be constructed using the dyadic *reshape* function signified by ρ:

```
      A←2 3ρ1 2 3 4 5 6      Form and store the matrix A.
      B←2 3ρ1 2 0 2 ¯1 0     Form and store the matrix B.
      A
1 2 3
4 5 6
      B
1 2 0
2 ¯1 0
```

The left argument of the reshape function in both of these cases is the vector 2 3; the 2 gives the number of rows and the 3 gives the number of columns. The number of elements given on the right side of ρ need not match the number of elements in the array. Extra elements will be ignored, or elements will be reused cyclically if needed to fill out the array. The first row is filled first, then the second row, and so on until the matrix is complete. Examples follow:

```
      2 5ρ1 2 3
1 2 3 1 2
3 1 2 3 1
```

```
      2 2ρ1 2 3 4 5 6 7
1 2
3 4
```

Vectors can also be constructed using the reshape function; in this case the left argument of ρ is the length of the vector being formed. For example,

```
      5ρ1
1 1 1 1 1
```

```
      5ρ4 2
4 2 4 2 4
```

```
      5ρ1 2 3 4 5 6 7
1 2 3 4 5
```

2.4 *Reduction*

In Chapter 1 reduction was used with vectors; for example +/1 4 5 is 1+4+5, which is 10. Reductions such as +/ or −/ also may take matrices as arguments. The result is a vector; each element of the vector is formed by "reducing" a row of the matrix. For example,

```
      B
1 2 3 4
0 1 1 1
2 1 0 1
```

```
      +/B
10 3 4
```

```
      -/B
¯2 ¯1 0
```

It is also possible to take a sum, product, or an alternating sum along the columns of a matrix instead of the rows. The symbol ⌿ is used to indicate that choice. With the same *B* you get

```
      +⌿B
3 4 4 6
```

```
      ×⌿B
0 2 0 4
```

```
      -⌿B
3 2 2 4
```

An example that uses reduction on matrices will be considered after a discussion of outer products.

2.5 Outer Product

The outer product operator, denoted by ∘ . and read "jot-dot", is used for generating arrays with "structure". If U and V are vectors and f is a dyadic scalar function, then $U \circ .fV$ is a matrix whose entry in the ith row and jth column is formed by applying f to the ith entry of U and the jth entry of V. For example, a times table can be produced by

```
      1 2 3 4 ∘.× 1 2 3 4 5 6
1  2  3  4  5  6
2  4  6  8 10 12
3  6  9 12 15 18
4  8 12 16 20 24
```

Other tables are constructed similarly:

```
      1 2 3 ∘.+ 1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

      1 2 3 ∘.* 1 2 3 4 5
1  1  1  1   1
2  4  8 16  32
3  9 27 81 243

      2 3 0 ∘.× 1 2 2
2  4  4
3  6  6
0  0  0

      ¯1 0 2 ∘.* 0 1 2 3
1 ¯1  1 ¯1
1  0  0  0
1  2  4  8
```

In general, the *outer product* $A \circ .fB$, where A and B are arrays, will be an array whose entries are formed by applying f with an element from A and an element from B in all possible pairs. Examples with a matrix argument are considered in the exercises.

2.6 Example: Evaluation of Polynomials at Several Points

Consider evaluating the polynomial $P(t) = 3 + 2t - 5t^2 + t^4$ for several values of t, say $t = 1, 3, 10$. Let `P←3 2 ¯5 0 1` be the vector of polynomial coefficients including a 0 for the t^3 term. You saw that `+/P×T*¯1+⍳5` would yield the value of the polynomial for a single value T; this expression, however, makes no sense if T is the vector `1 3 10`. Powers of each of the T values of interest are required. The outer product makes the appropriate "power table". For example,

```
      T←1 3 10
      T∘.*¯1+⍳5
1     1     1     1     1
1     3     9    27    81
1    10   100  1000 10000
```

Get a power table.

```
      3 5⍴P
3 2 ¯5 0 1
3 2 ¯5 0 1
3 2 ¯5 0 1
```

Get multiple copies of the polynomial coefficients.

```
      (3 5⍴P)×T∘.*¯1+⍳5          Multiply elementwise to get "terms".
3      2    ¯5     0      1
3      6   ¯45     0     81
3     20  ¯500     0  10000

      +/(3 5⍴P)×T∘.*¯1+⍳5        Add along the rows.
1    45   9523

      T←3 5 10 ¯2 0
      +/(5 5⍴P)×T∘.*¯+⍳5
45   513 9523     3     3
```

Notice that any number of T values could have been used. In Section 4.7 the matrix product is used in an expression for the evaluation of a polynomial at several points, which avoids the reshape of P required here. Finally, in Section 9.14 polynomial evaluation is accomplished with the APL primitive function base value.

2.7 Catenation

The APL function *catenate,* designated with a comma, is used to attach arrays together. It can be used to attach a scalar to a scalar, a scalar to a vector, and a vector to a vector. The result in each of these uses is a vector:

```
      1 2,⍳3              1,2,3 4,5 6          1,¯4,5
1 2 1 2 3           1 2 3 4 5 6          1 ¯4 ¯5
```

Catenation may be used with reshape to construct a matrix from vectors. Given vectors U, V, W each with five entries, the matrix M having those vectors as rows can be formed by

```
      U
1 2 3 4 5

      V
2 2 2 2 2
      W
6 5 3 1 1
```

```
      M←3 5⍴U,V,W

      M
1 2 3 4 5
2 2 2 2 2
6 5 3 1 1
```

2.8 Catenation of Matrices

Matrices can also be catenated. With matrices you need to indicate whether the matrices are to be attached side by side or one above the other. The comma in the expression $A\ ,\ B$ indicates the matrices A and B are to be attached side by side. A comma followed by a 1 in square brackets as in $B\ ,[1]\ C$ indicates the matrices are to be attached B above C. The [1] indicates attachment along the "first axis". Alternately, a comma overstruck with a minus sign as in $A\ \overset{-}{,}\ B$ may be used on many systems to indicate catenation along the first axis, and $A\ ,[2]\ B$ may be used to indicate catenation along the second axis. See the exercises for an additional discussion of selecting the axis of application of a function. Examples follow:

```
      A
1 2 3
4 5 6
7 8 9

      B
1 0
0 1
0 0

      C
1 2
7 5

      A,B
1 2 3 1 0
4 5 6 0 1
7 8 9 0 0

      A,3
1 2 3 3
4 5 6 3
7 8 9 3

      B,[1]C
1 0
0 1
0 0
1 2
7 5

      A,[2]B
1 2 3 1 0
4 5 6 0 1
7 8 9 0 0

      A,[1]3
1 2 3
4 5 6
7 8 9
3 3 3

      B⍪C
1 0
0 1
0 0
1 2
7 5
```

Notice that in `A,3` the scalar 3 was automatically extended to be a 3-by-1 matrix so that the catenation made sense. Catenation requires matching lengths of the edges along which attachment occurs. For example, `A,C` results in an error message.

2.9 *Example: Forming a Blockwise Diagonal Matrix*

Catenation is convenient for constructing matrices with certain types of structure. In order to produce the matrix

```
1 2 0 0
3 4 0 0
0 0 1 0
0 0 0 1
```

these intermediate matrices `A`, `B` and `Z` may be formed:

```
      A←2 2⍴⍳4
      B←2 2⍴1 0 0
      Z←2 2⍴0

      A
1 2
3 4

      B
1 0
0 1

      Z
0 0
0 0

      A,Z
1 2 0 0
3 4 0 0

      Z,B
0 0 1 0
0 0 0 1

      (A,Z),[1]Z,B
1 2 0 0
3 4 0 0
0 0 1 0
0 0 0 1
```

2.10 Catenation of Matrices with Vectors

It is often convenient to catenate a vector to a matrix, as in augmenting a table with a new row or column of data. This is always possible by first reshaping the vector into a matrix and using catenation as previously discussed. Most modern versions of APL allow a vector to be catenated directly to a matrix, thereby simplifying many array constructions.

For a vector to be catenated as a column to a matrix, the length of the vector must match the number of rows in the matrix. It is similar for catenation of a vector as a row above or below a matrix. Here `A , U` and `A , [2] U` indicate `U` is catenated beside `A`, and `A , [1] U` and `A ⍪ U` indicate `U` is to be attached beneath `A`. Consider an example with a matrix `A` and vectors `U` and `V`:

```
      A
1 2 3
4 5 6

      U
8 8

      V
13 15 18

      A,U
1 2 3 8
4 5 6 8

      A,[2]U
1 2 3
4 5 6

      U,A
8 1 2 3
8 4 5 6

      A,[1]V
 1  2  3
 4  5  6
13 15 18

      A⍪V
 1  2  3
 4  5  6
13 15 18

      V,[1]A
13 15 18
 1  2  3
 4  5  6
```

With `A` and `V` as above, the expression `A , V` would result in an error message since the number of entries in `V` is not the same as the number of rows of `A`.

2.11 Indexing

Entries or blocks of entries of an array can be referenced by using "position" numbers or *indices* to specify entries. For vectors, you list the indices of the desired elements in their desired order inside square brackets. Consider these examples with the vector `C`:

```
      C
¯1 3 ¯5 6 4

      C[2]
3

      C[1 2 3]
¯1 3 ¯5

      C[2 1 5]
3 ¯1 4

      C[4 4 1]
6 6 ¯1

      C[2 2⍴1 2 5 3]
¯1  3
 4 ¯5
```

The last example above uses a matrix of indices.

To "index a matrix," you list the desired row and column indices, separated by a semicolon, inside square brackets. The result is the array with the entries from the matrix arising from ***all combina-***

tions of each row index with each column index, as follows:

```
      D
1 1 0 2
1 1 1 4
1 2 1 3

      D[2;4]
4

      D[3;4 2 1]
3 2 1

      D[1 2;2 3]
1 0
1 1
```

If an index is omitted, ***all*** the row indices or column indices are assumed. This gives a convenient notation for referring to rows and columns of a matrix. For example,

```
      D[1;]
1 1 0 2

      D[2 1;]
1 1 1 4
1 1 0 2

      D[;4]
2 4 3
```

The result of each of `D[1;]` and `D[;4]` is a ***vector;*** in particular, `D[1;]` is not a row matrix and `D[;4]` is not a column matrix.

A matrix can be modified by assignment using indexing as follows:

```
      D
1 1 0 2
1 1 1 4
1 2 1 3
      D[2;2]←5
      D
1 1 0 2
1 5 1 4
1 2 1 3
      D[1;]←1 2 3 4
      D
1 2 3 4
1 5 1 4
1 2 1 3
      D[1 2;1 2]←2 2⍴5 6 7 8
      D
5 6 3 4
7 8 1 4
1 2 1 3
```

2.12 Example: Elementary Row Operations on Matrices

The reduction of a matrix to an echelon form by use of elementary row operations has several applications (see an introductory linear algebra text). The row operations are:

I. Add a multiple of one row to another.
II. Multiply a row by a nonzero constant.
III. Interchange two rows.

These are easily accomplished. For example,

```
      A
1 1 0 2
1 1 1 4
2 4 2 6
      A[2;]←A[2;]+¯1×A[1;]
      A
1 1 0 2
0 0 1 2
2 4 2 6
      A[3;]←A[3;]+¯2×A[1;]
      A
1 1 0 2
0 0 1 2
0 2 2 2
      A[2 3;]←A[3 2;]
      A
1 1 0 2
0 2 2 2
0 0 1 2
      A[2;]←.5×A[2;]
      A
1 1 0 2
0 1 1 1
0 0 1 2
      A[2;]←A[2;]-A[3;]
      A
1  1  0  2
0  1  0 ¯1
0  0  1  2
      A[1;]←A[1;]-A[2;]
      A
1  0  0  3
0  1  0 ¯1
0  0  1  2
```

The first row times ¯1 is added to the second row.

The first row times ¯2 is added to the third row.

Rows 2 and 3 are interchanged.

Multiply the second row by .5.

A is in reduced echelon form.

2.13 Shape

Recall that the length of a vector *V* is given by ⍴*V*. In general the shape of an array can be determined using the monadic *shape* function denoted by ⍴; for example,

```
      A
1 1 0 2
1 1 1 4
1 2 1 3

      ρA
3 4

      +/A
4 7 7

      ρ+/A
3

      ρA[2 3;1 2 3]
2 3
```

```
      B
1 2 ¯3 2 1

      ρB
5

      B[1]
1

      ρB[1]
```

(The result is "empty".)

```
      ρB[2 3]
2
```

Notice that A and $A[2\ 3;1\ 2\ 3]$ are matrices and their shape has two components. The expressions $+/A$ and B and $B[2\ 3]$ represent vectors, so their shapes each have one component. $B[1]$ is a scalar, so its shape is the *empty vector*. In general, the shape of an array is the vector whose elements give the upper limits on the indices of the array. The array A has shape $3\ 4$, which means its first index ranges from 1 to 3 and its second index ranges from 1 to 4. You can also say the length of the *first axis* of A is 3 and the length of the *second axis* of A is 4.

The shape of the shape of an array as in $\rho\rho A$ is the number of indices; this is called the *rank* or *dimension* of the array. For the same A and B above, you have

```
      ρA
3 4

      ρρA
2
```

```
      ρB
5

      ρρB
1
```

```
      ρB[1]
(Empty.)

      ρρB[1]
0
```

The dimension of a matrix is 2; the dimension of a vector is 1; the dimension of a scalar is 0. Consider the differences among a column matrix, a row matrix, and a vector:

```
      A←3 1ρ3 5 7

      A
3
5
7

      ρA
3 1

      ρρA
2
```

```
      B←1 3ρ3 5 7

      B
3 5 7

      ρB
1 3

      ρρB
2
```

```
      C←3 5 7

      C
3 5 7

      ρC
3

      ρρC
1
```

The arrays *B* and *C* appear to be the same when they are displayed; their shapes and dimensions, however, are different. The shape of an array is not always explicitly declared; ***when using APL you can make subtle errors by incorrectly assuming that an array has a shape different from what it actually has. Check the shape and the dimension of APL objects when errors occur.***

2.14 Empty Arrays

In the discussion of shape it was observed that scalars are arrays with "empty" shape and dimension 0. The shape of a scalar, although empty, is a legitimate APL array that may itself be manipulated. Here are two more examples of empty arrays; they are created by reshape and index generator with 0 argument:

```
      2⍴5          1⍴5          0⍴5
5 5            5              (Empty.)

      ⍳2           ⍳1           ⍳0
1 2            1              (Empty.)
```

Notice the appropriateness of the result of `0⍴5` in the light of the results for the diminishing arguments; similarly for `⍳0`. This is illustrative of the internal consistency of APL.

The empty arrays considered so far are examples of an empty vector—that is, a vector with zero elements. Notice below that `⍴⍴A` is `1`, indicating that *A* really is a one-dimensional array—namely, a vector. These empty vectors can be manipulated as vectors. Sometimes this results in an empty vector and sometimes not; for example,

```
      A←⍴3         B←0⍴5        C←⍳0

      ⍴A           ⍴B           ⍴C
0              0              0

      ⍴⍴A          ⍴⍴B          ⍴⍴C
1              1              1

      A,2 3 4      A+5          A,5
2 3 4          (Empty.)       5
```

Matrices may also be empty in the sense that one or both of the axes may have length 0; see Exercise 23. Empty arrays are useful for initializing arrays to be constructed iteratively.

Exercises

1. What are the results of the following expressions?

(a) `2 3⍴1 2 6 3` **(b)** `2 3⍴1 2 3 4 5 6 7 8 9`
(c) `5⍴1 2 3` **(d)** `5⍴1 2 1 2 3 4 3 4 5 6`

2. Let `A←2 3⍴⍳6` and `B←2 3⍴1 2`. Use APL to find the following:

(a) `A+B` **(b)** `A÷B`
(c) `A*2` **(d)** `3+B`

(e) `A=B` (f) `B=2`
(g) `2*A`

3. Write APL expressions to form these matrices:

(a)
```
1 2 3 4
1 2 3 4
1 2 3 4
```
(b)
```
1 2 3 1
2 3 1 2
3 1 2 3
```
(c)
```
1 1
1 1
1 1
```
(d)
```
1
2
3
```

4. Give a brief expression for the sum of all the elements of a matrix `A`. Give another one.
5. The axis of an array along which a reduction "operates" can be explicitly selected. Namely, $f/[i]$ indicates an f-reduction along the ith axis. Let `A←2 3⍴1 2`. Find these arrays:

(a) `+/A` (b) `+⌿A`
(c) `+/[1]A` (d) `+/[2]A`
(e) `+⌿[1]A` (f) `+⌿[2]A`

6. Write APL expressions using the outer product operator to form these matrices:

(a)
```
1  2  4
3  6 12
4  8 16
```
(b)
```
2 3 4 5
3 4 5 6
4 5 6 7
```
(c)
```
1 2  4   8
1 5 25 125
1 3  9  27
```

7. Give an expression for evaluating $P(t) = 5 + 6t + {}^{-}4t^4 + t^5$ at the points $t = 1, 3, 5, 7, 9$.
8. What is the result of

(a) `2,3,¯2` (b) `2,-3,4`
(c) `-2,3,-4` (d) `6⍴2,3`
(e) `6⍴2 3` (f) `6,2⍴3`
(g) `6 2⍴3`

9. What is the result of

(a) `6⍴0` (b) `1,6⍴0` (c) `6 6⍴1,6⍴0`

10. Let `A←2 3⍴1` and `B←2 3⍴2 3 4`. What is the result of

(a) `A,B` (b) `A,[1]B` (c) `A,[2]B` (d) `A⍪B`.

11. Let `A←2 3⍴1` and `B←2 2⍴2`. Observe the result or the error message:

(a) `A,B` (b) `A,[1]B` (c) `A,[2]B` (d) `A⍪B`.

12. Use catenate to construct the matrices from obvious simpler matrices:

(a)
```
1 2 3 3
3 4 3 3
3 3 1 2
3 3 3 4
```
(b)
```
1 1 1 1 2 3
0 0 0 4 5 6
1 1 1 1 1 1
0 0 0 0 0 0
```

13. Let `A←2 3⍴4 2 5 0 1 2` and `U←¯4 3 1`. Predict the results of the following expressions:

(a) `A,[1]U` (b) `U,[1]A`
(c) `A,2 3` (d) `3 7,A`
(e) `A,3` (f) `A,U`

14. *Higher dimensional* arrays are available in APL. Many of the functions and operators extend to such arrays. Consider the three-dimensional array `A←2 3 4⍴⍳24`. Determine what the following expressions yield, and then use APL to check your expectations. How does APL display a three-dimensional array?

(a) `A` **(b)** `⍴A` **(c)** `⍴⍴A`

Indexing requires three indexes:

(d) `A[1;1;2]` **(e)** `A[1;2;1]` **(f)** `A[2;1;1]`
(g) `A[1;1;]` **(h)** `A[1;;1]` **(i)** `A[;1;1]`
(j) `A[1;;]` **(k)** `A[;1;]` **(l)** `A[;;1]`

Reduction can be along any of three axes; they can be explicitly selected via `/[i]`:

(m) `+/[1]A` **(n)** `+/[2]A` **(o)** `+/[3]A`
(p) `+/A` **(q)** `+⌿A`

Notice that $f/$ indicates reduction along the last axis, and that $f⌿$ indicates reduction along the first axis. Catenation can be along any of three axes:

(r) `A,[1]3` **(s)** `A,[2]3` **(t)** `A,[3]3`
(u) `A,3` **(v)** `A⍪A`

Notice that catenation without axis specification is along the last axis, and catenation indicated by `⍪` is along the first axis.

15. Produce a three-dimensional array via outer products as follows: let `B←⍳4` and `C←2 3⍴⍳6` and `A←B∘.×C`. Then redo (a)–(v) of Exercise 14.

16. Let `A←3 4⍴⍳12`. Find the result of

(a) `A[3;2]` **(b)** `A[3;2 3]`
(c) `A[1 2;3 4]` **(d)** `A[2 1;3 4]`
(e) `A[2;]` **(f)** `A[;2]`
(g) `A[2 3;]` **(h)** `A[3 2;]`

17. Use elementary row operations to put the following matrices into reduced echelon form:

(a)
```
1  ¯2  2  5
1   1  1  4
2   0  1  2
```
(b)
```
2  ¯4  1   0  3
0  ¯4  1  ¯1  0
1  ¯2  1   1  5
```
(c)
```
2   1   2   5
1   1   2   2
2  ¯1  ¯2   2
```

18. Let `A←2 3⍴1 0 3 4 2 1` and `B←2 2⍴⍳4`. Observe the results or the error messages produced by

(a) `A,[1]B` **(b)** `A,[2]B` **(c)** `A,[3]B`
(d) `A⍪B` **(e)** `÷/A` **(f)** `÷⌿A`

19. Suppose `⍴A` is `3 4` and `⍴B` is `4 4`. What are the shapes of the following arrays?

(a) `A,[1]B` **(b)** `A*2` **(c)** `A[1 2;]`
(d) `B,B` **(e)** `A[1;]` **(f)** `B[;2 3⍴⍳4]`
(g) `+/A` **(h)** `B[1;2]` **(i)** `A[2 3;1 2]`
(j) `A[2;2 2⍴⍳4]` **(k)** `A[1;]∘.×A[;1]` **(l)** `+/B,B`

(Suggestion: Choose specific A and B and experiment.)

20. Let *V* be a vector and *I* be a matrix of indices for *V*. How is the shape of *V*[*I*] related to the shape of *I* and *V*?

21. Predict the result of the following APL expressions. Some are empty. Check your work. Let *B*←0ρ2 and *C*←⍳0.

(a) *B*,3 4 **(b)** *B*,3
(c) *B*,*C* **(d)** *B*,ρ*B*
(e) *B*+*C* **(f)** *B*÷0
(g) *B*=*C*

22. Predict the results of the following:

(a) +/3ρ5 **(b)** +/2ρ5 **(c)** +/1ρ5 **(d)** +/0ρ5 **(e)** +/⍳0
(f) ×/3ρ5 **(g)** ×/2ρ5 **(h)** ×/1ρ5 **(i)** ×/0ρ5 **(j)** ×/ρ5
(k) ÷/3ρ5 **(l)** ÷/2ρ5 **(m)** ÷/1ρ5 **(n)** ÷/0ρ5 **(o)** ÷/3+ρ5

23. Predict the result of the following APL expressions. Some are empty. Indicate the shape and dimension of the empty arrays. Check your work. Let *E*←3 0ρ¯2 and *F*←0 4ρ1 and *G*←0 0ρ1.

(a) *E*,2 **(b)** *E*,1 2 5 **(c)** *E*,*E*
(d) *E*,[1]8 **(e)** *E*,[2]8 **(f)** *F*,4
(g) *F*=1 **(h)** *G*,1 **(i)** (*G*,1),[1]3
(j) *G*÷0

3

Elementary Function Definition

User-defined APL functions will allow effective treatment of complicated problems. The interactive environment of APL will remain, but it will be enhanced by the defined functions. This chapter deals with functions with explicit arguments, explicit result, and at most a simple loop. It discusses editing of user-defined functions and relevant system commands and introduces random number generation. As examples of functions using simple loops, the chapter includes Newton's method and the computation of the Fibonacci numbers.

3.1 Function Definition

Each primitive APL function conforms to the mathematical notion of a function; each has one or two arguments and generates a uniquely determined value, which is called the result of the function. One or both of the arguments and the result of these functions are often arrays comprised of many components. Moreover, the result from any primitive function is passed on as an argument for the next function to the left. These properties of the primitive functions are shared by the types of user-defined functions treated in this chapter.

The del symbol, ∇, is used to signal APL to begin function definition mode. You begin the definition of a function with a "header", which includes the function name and variable names for the result of the function and its arguments. For example, to define a monadic function to be named *FCN* with result *Y* and argument *X*, you would type

```
∇Y←FCN X
```

Note that the argument for a monadic function is to the right of the function name. After you type the header, APL will prompt you for the first line of the function with [1] at the left margin:

```
∇Y←FCN X
[1]
```

After you enter each line, APL prompts with the next higher line number. Several lines may be entered; another ∇ is used to indicate that the function definition is complete.

Here is a specific example. A monadic function named *AVG* is defined. It has a vector argument named *V*; it computes the average of the entries of *V*, and the function result is named *Z*:

```
      ∇Z←AVG V
[1]   Z←(+/V)÷⍴V
[2]   ∇
```

The formula for the computation forms the "body" of the function. The assignment "*Z*←" in the header and in line [1] is essential to the automatic passing of the result of the function to another function.

After the closing ∇ is entered APL resumes immediate execution mode, and the function *AVG* can be used simply by referring to it by name. A space is required between the function name and its argument:

```
      AVG 1 3 4          AVG 6 7 11          6 7 11 - AVG 6 7 11
2.666666667        8                    ¯2 ¯1 3
```

Notice that the result of *AVG* 6 7 11 — namely 8, could be used in the computation of differences in 6 7 11 − *AVG* 6 7 11. A function is said to "produce a result" or "return a result" when the result of its computation is available for further computation. The variables *V* and *Z* used in the definition of *AVG* are *localized* in the sense that their use within the function *AVG* will not conflict with values assigned to the variables *V* or *Z* in the context in which the function is called. Here *AVG* was called from the active workspace, and hence any values of *V* or *Z* in the active workspace would not have changed. Moreover, if *V* and *Z* did not exist in the active workspace, they would still not exist after execution of *AVG*.

Next is an example of a dyadic function. Note particularly the placement of the variable names for the arguments in the header. The function is named *DOT*; it has two vector arguments named *U* and *V*, and the result, which is the dot (scalar) product, is named *Z*:

```
      ∇Z←U DOT V
[1]   Z←+/U×V
[2]   ∇

      1 0 DOT 0 1                    1 0 2 DOT 1 2 3
0                                7

      R←1 2 DOT 3 4                  2×1 2 DOT 3 4
                                 22

      5+R
16
```

The variables *U*, *V*, and *Z* are local to the function *DOT*. In the fourth illustration the result of the *DOT* function was available as an argument of the × function. The result of the function was assignable to a variable.

Observe the independence of the variables *U* and *V* in the workspace from localized variables with the same name in the function *DOT*:

```
      U←1 1 2
      V←2 3 1
      1 2 DOT 3 4
11
      U
1 1 2
      V
2 3 1
      U DOT U
6
      U
1 1 2
      V
2 3 1
```

You may want to preview the material on editing and suspended functions in Sections 3.5 to 3.8 before experimenting with defined functions.

3.2 Example: A Function for Polynomial Evaluation

The polynomial $P(t) = a_0 + a_1t + a_2t^2 + \cdots + a_nt^n$ can be represented by the vector $(a_0, a_1, a_2, \ldots, a_n)$, which lists its coefficients including zero coefficients. Thus, $P(t) = 3 - 4t + 2t^3$ corresponds to `3 ¯4 0 2`. A slight modification of the example of polynomial evaluation given in Section 2.1 gives the defined function *POLYVAL* with the polynomial coefficients *P* as the left argument and the value *T* at which it is to be evaluated as the right argument. Note that `¯1+⍳⍴P` gives the vector of powers to which *T* is to be raised:

```
      ∇ Z←P POLYVAL T
[1]   Z←+/P×T*¯1+⍳⍴P ∇          The closing ∇ may end a line.
```

A few values of the polynomials $1 + 2t + t^2$ and $3 - 4t + 2t^3$ are computed:

```
      1 2 1 POLYVAL 1                    1 2 1 POLYVAL ¯1
4                                  0

      3 ¯4 0 2 POLYVAL 3                 3 ¯4 0 2 POLYVAL 0
45                                 3
```

3.3 Example: Approximating Areas Using Right-Hand Endpoints

A user-defined function may use the results of other user-defined functions that produce explicit results simply by referring to them by name within the body of the function. Here a function, *RHEP*, which approximates the area under a curve, will use another function, *F*, which gives the y coordinates of points on the curve.

To approximate the area under the graph of a nonnegative function $y = f(x)$ and above the interval $[a, b]$, the areas of n rectangles associated with the curve are added together (Figure 3.1). Each rectangle has width $\Delta x = (b - a)/n$. The height of the kth rectangle is $f(x_k)$, where $x_k = a + k\Delta x$; thus

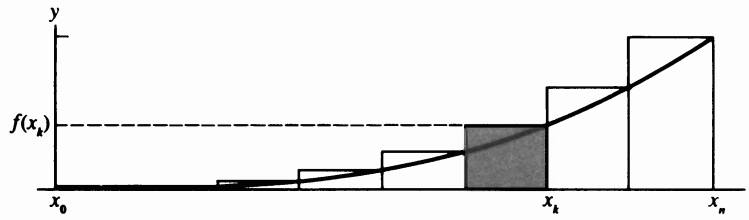


Figure 3.1 Approximation of the area under $y = x^3$ and above $[0, 1]$ using $n = 8$ rectangles based at the right-hand endpoint.

the total area is approximated by

$$A \approx \sum_{k=1}^{n} f(x_k)\,\Delta x = \Delta x \sum_{k=1}^{n} f(x_k)$$

A defined function, *RHEP*, that computes that approximation is given. *RHEP* has as arguments the number *N* of rectangles and the vector *INT* that has the endpoints of the interval $[a, b]$ as its components. The function *F* is defined separately and is chosen to be the cubing function in this example. Notice that on line 5 the function *F* is invoked simply by using its name.

```
      ∇ Z←N RHEP INT
[1]   A←INT[1]
[2]   B←INT[2]
[3]   DX←(B-A)÷N          The width of each rectangle.
[4]   X←A+DX×⍳N           Vector of subinterval right-hand endpoints.
[5]   Y←F X               Function values at right-hand endpoints.
[6]   Z←DX×+/Y ∇          Add function values and multiply by width.

      ∇ Z←F T
[1]   Z←T*3 ∇
```

Here are some sample approximations:

```
      10 RHEP 0 1                    100 RHEP 0 1
0.3025                         0.255025

      1000 RHEP 0 1                  1000 RHEP 1 3
0.25050025                     20.026008
```

The variables in the header of the function, *N*, *INT*, and *Z* are automatically localized, but the variables *A*, *B*, *DX*, *X*, and *Y* are global in the sense that they will replace variables by those names in the context in which *RHEP* was called. Here *RHEP* is called from the active workspace, and hence the active workspace contains the values of *A*, *B*, *DX*, *X*, and *Y* defined in *RHEP*. For example,

```
      A←N←12               Assign values to global variables A and N.
      10 RHEP 0 1
0.3025
      A                    A in RHEP is global, hence A is changed.
0
      N                    N in RHEP is local, hence N is preserved.
12
```

The variables used by *RHEP* can be localized simply by listing them on the header line with semicolons separating distinct variables. All variables in the function *RHEP2* are localized:

```
      ∇ Z←N RHEP2 INT;A;B;DX;X;Y
[1]   A←INT[1]
[2]   B←INT[2]
[3]   DX←(B-A)÷N
[4]   X←A+DX×⍳N
[5]   Y←F X
[6]   Z←DX×+/Y ∇

      A←N←12
      10 RHEP2 0 1
0.3025
      A      A in RHEP2 is local, hence A is preserved.
12
      N
12
```

3.4 Random Integers, Random Reals

APL provides a simple way to generate random numbers. The question mark is used to denote the *roll* function. A random integer between 1 and *N*, where *N* is a positive integer is generated by ?*N*. Roll is a monadic scalar function. Examples are

```
      ?100                ?100 200 200 100          ?4⍴10
14                  76 92 107 22              1 7 7 10
      ?6 6 6 6            ?1E17                     1E¯17×?1E17
3 4 5 1             5.346163529E15            0.6711493841
```

A random real number from [0, 1] can be computed by generating a random integer up to 1*E*17 and then multiplying by 1*E*¯17. The magnitude 1*E*17 is used because it is on the order of the precision of the real numbers on most APL systems. The function *RANREAL* produces *N* random real numbers between 0 and 1:

```
      ∇ Z←RANREAL N
[1]   Z←1E¯17×?N⍴1E17 ∇

      RANREAL 3
0.6711493841 0.3834156508 0.4174859748
```

3.5 Correcting a Function Line

A previously defined function can be accessed for modification by typing ∇ followed by the function name. The line number of the line to be modified can be typed in brackets, [*N*], and the line can be retyped. Other lines can be changed or added, and definition mode is closed with another ∇.

Consider the function *RAND*, which is intended to select randomly *N* of the numbers 0 1 2 3 4 5 with replacement. A first test of the function might yield the indicated result:

```
      ∇Z←RAND N
[1]   Z←?N⍴5 ∇

      RAND 30
3 5 5 3 1 4 3 4 5 4 2 1 4 2 4 4 5 2 2 5 4 4 4 1 4 5 2 3 4 3
```

The testing produced no zeros. That suggests a problem with the definition. *RAND* can be corrected as follows:

```
      ∇RAND          Use the function name, not the header.
[2]   [1]            Get to line [1].
[1]   Z←¯1+?N⍴6      Replacement line is entered.
[2]   ∇              Close definition mode.
```

The function editing begins when you type ∇*RAND*. APL responds with a prompt for a line at the end of the function, here [2]. Since you need to replace line [1], you type [1]; APL responds with [1]. Then APL awaits the new line [1]. You enter the replacement for line [1], *Z*←¯1+?*N*⍴6, and APL prompts for the next line—namely, [2]. The closing ∇ ends definition mode. Testing the corrected function yields

```
      RAND 30
1 1 2 0 2 5 5 0 5 3 3 1 5 2 1 0 5 0 3 2 1 5 3 2 5 0 4 4 4 0
```

3.6 Commands for Editing Functions

Functions that have been defined can be edited easily. The command ∇*FCT* must be used first to begin (open) function definition mode in order to modify the function called *FCT*. Notice that the function name and not the header is used. All of the following commands except opening definition mode are made in response to an APL function definition mode prompt that is of the form [*L*], where *L* is a line number. Press ⟨Return⟩ to complete each command. The last command, a closing ∇, is used to end (close) definition mode and return to immediate execution mode.

The commands for editing functions follow:

∇*FCT*	Opens *FCT* for editing and prompts the user with a line number one higher than the last line in the function.
[□]	Displays the current definition of *FCT* and prompts the user for a line at the end of the function.
XXX	If the expression *XXX* is typed in response to the APL definition mode prompt [*L*], *XXX* replaces the content of line *L*.
[*N*]	APL prompts for replacement or the addition of line *N*, where *N* may be a decimal fraction—for example, [2.3] inserts this line in order between the appropriate lines.
[0]	APL prompts for replacement of the header.

[*N*] *XXX*	Modifies line *N* to be the expression *XXX*.
[*N*□0]	Displays line *N* for modification with arrow keys, backspace, and by inserting and deleting characters.
[*N*□*M*]	Begins a two-pass editing mode. This is archaic but useful on hardcopy terminals that do not support some of the other commands. First, line *N* is displayed and the carriage is moved to column *M* on the next line. On this pass, under the displayed line the user may put a slash under characters to be deleted and/or put a digit for the number of spaces to be inserted before the position of the digit. On the next pass, the modified line is displayed and the user may type characters into the blank spaces.
[∆*N*]	Delete line *N*.
∇	Close definition mode; lines are renumbered to consecutive integers starting with 1.

Some of these commands may be combined into a single expression. Examples of this are

∇*FCT*[□]	Open *FCT* for editing; display the entire function; prompt for adding lines at the end of the function.
∇*FCT*[□]∇	Display the function *FCT* and return to immediate execution mode.

For example, in order to display the function *RAND* from Section 3.5 enter

```
      ∇RAND[□]∇
    ∇ RAND N
[1]   Z←¯1+?Nρ6
    ∇
```

This function display format is used in this book.

Full screen editors are available on many modern APL systems. You will find it useful to investigate your system's screen editing capabilities.

3.7 *Suspended Functions; Interrupts*

When a user-defined function halts in midcomputation due to an error, APL returns to immediate execution mode in a modified state; namely, the function execution is *suspended*. Sometimes several functions have their execution suspended at the same time. Basically APL is ready for modifications of functions and variables and a resumption of the computation. In Appendix A the modified state caused by suspended functions will be discussed and used for debugging. Until then, since there is a small possibility that a suspended function would interfere with future work, it is best to clear the suspension of the function with the system command)*RESET* immediately after the suspension occurs. The suspended functions may be listed with the state indicator command)*SI*.

You may also interrupt computations, thereby producing function suspension. To signal an interruption at the next line (weak interrupt), you strike the break or attention key once. For an interruption at the next function evaluation (strong interrupt), strike it twice. On some keyboards there is a separate

key marked attention ⟨attn⟩ or break ⟨break⟩. Often you must hold down the control key while you strike the break key ⟨ctrl-break⟩ in order to initiate an interrupt. Here is an example of a user interrupt and the use of) *RESET* to clear the state indicator:

```
      APPROX←10000 RHEP 0 1      Computation requested.
      ⟨ctrl-break⟩⟨ctrl-break⟩   Strong interrupt requested.
INTERRUPT
RHEP[4] X←A+DX×⍟N
                 ^                Place of interrupt shown.
      )SI
RHEP[4] *                        Execution of RHEP is suspended.
      )RESET                     Clears the state indicator.
      )SI
                                 No suspended functions remain.
```

3.8 System Commands

A few system commands helpful for managing workspaces are noted here. When you have defined several functions, it is convenient to save those definitions for later use. See Appendix B for a complete set of system commands and more examples.

) *SAVE n W* Saves the active workspace with name *W* on account number *n*. The workspace includes all variables and user-defined functions. Previous workspaces with the same name and account number are overwritten. The account number is optional. On some microcomputer systems, the account numbers 1, 2, 3, and so on, correspond to disk drives A:, B:, C:, and so on, with various conventions. APL responds with some information such as the time of the save.

) *LOAD n W* Loads the workspace with name *W* from account number *n*. See above remarks about account numbers. APL responds with some information such as the time the workspace was last saved.

) *CLEAR* Loads a clear workspace. System variables are restored to their default values. All user-defined functions and variables are lost.

) *ERASE XXX* Erases functions and variables listed in *XXX*.

) *FNS* Lists the defined functions in the active workspace.

) *VARS* Lists the defined variables in the active workspace.

) *SI* State indicator; lists the suspended functions.

) *RESET* All functions are reset to a state of nonsuspension. The state indicator is cleared.

If you defined all the functions in this chapter so far, you could display the function names and save the workspace with the name *CHAP3* as shown. Then *POLYVAL* and *RANREAL* are erased, the ac-

tive workspace is cleared, and finally the workspace *CHAP3* is restored with the) *LOAD* command:

```
      )FNS
AVG     DOT      F        POLYVAL  RAND      RANREAL  RHEP
      )SAVE CHAP3
CHAP3 SAVED 1988-07-12 15:34:25
      )ERASE RANREAL POLYVAL
      )FNS
AVG     DOT      F        RAND     RHEP
      )CLEAR
      )FNS
                                  (No user-defined functions are in a clear workspace.)
      )LOAD CHAP3
SAVED 1988-07-12 15:34:25
      )FNS
AVG     DOT      F        POLYVAL  RAND      RANREAL  RHEP
```

Note that the) *ERASE* and) *CLEAR* commands applied only in the active workspace; the saved workspace *CHAP3* was unaffected by those commands.

3.9 Branching: Simple Loops

The right pointing arrow, →, is used monadically to branch to a given line. That is, →*N* indicates that the next line to be executed in a defined function is line number *N*; that expression is read "go to *N*" or "branch to *N*". Branching is used in limited ways throughout the text and is treated in greater detail in Chapter 10. Only simple loops are considered here. Often, good APL functions avoid loops that would be used in other programming languages. Until Chapter 10, it will be noted whenever a loop or branching is appropriate in the solution of an exercise.

The simple loops considered here will be of a particular form: Initial data is given and then a formula is iterated several times. When the loop has been repeated sufficiently many times, the execution of the function terminates.

It is convenient to use the *less than* function of APL for these loops. Like equality, less than results in a 1 or a 0, depending on whether the inequality is true or not:

```
      2<3          3<2          2=2
1                  0            1

      2<2          3<¯2 5       2=1
0                  0 1          0

      4×2<3        4×3<3        4×5<3
4                  0            0
```

Notice that 4×*K*<*N*, for scalars *K* and *N*, results in either a 4 or a 0, depending on whether the inequality is true or not.

Consider the simple case of computing the factorial function using a loop. The factorial function in common notation is $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$, with the convention that $0! = 1$. At each

execution of line 5 of *FAC*, one of the multiplications indicated is effected. On line 3, the expression →4×*K*<*N* causes a branch to line 4 or to 0. A branch to 0 causes the function to end execution and return the current value of *Z*. Line 6 causes a branch to line 3:

```
     ∇ Z←FAC N;K
[1]    Z←1          The result if N=0.
[2]    K←0          Set iteration counter to 0.
[3]    →4×K<N       Go to 4 if K<N or go to 0 (exit) if K=N.
[4]    K←K+1        Increment the iteration counter.
[5]    Z←Z×K        Multiply by the iteration count.
[6]    →3           Branch to 3.
     ∇
      FAC 0              FAC 1              FAC 2
1                  1                  2

      FAC 4              FAC 6              FAC 10
24                 720                3628800
```

Notice that the main lines 4 and 5 are repeated while *K*<*N* and that each time lines 4 and 5 are executed the value of *Z* is replaced by *Z* times the iteration count. Although this loop is instructive of branching, the factorial function can be computed easily via ×/⍳*N*, and, moreover, there is a primitive APL function for factorial, which will be introduced in Section 6.5.

3.10 Example: Newton's Method

Newton's method is a technique for numerically approximating solutions to an equation $f(x) = 0$. It is based upon the idea that an estimate for a root determines a tangent line on the graph of $y = f(x)$, and the next estimate can be taken to be the point where that tangent line intersects the x axis. Normally this procedure is iterated several times. When Newton's method works, it tends to work very well. See a calculus or numerical analysis text for the details and a discussion of the limitations of Newton's method.

The computation used in Newton's method gives a new estimate for a root x_{n+1} in terms of the current estimate for the root x_n (Figure 3.2).

Given APL functions *F* and *DF* that compute $f(x)$ and $f'(x)$, Newton's method can be iterated *N* times to approximate a solution to $f(x) = 0$ with the function *I TERNEWT*. Here $\sqrt{2}$ is approximated

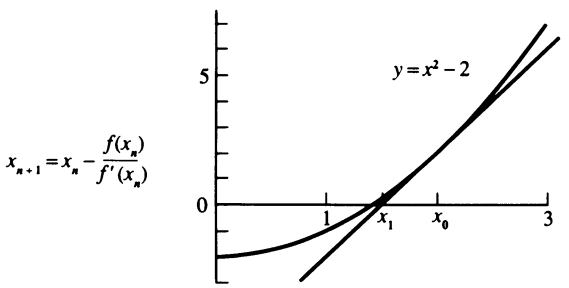


Figure 3.2 Newton's method.

by finding a root of $f(x) = x^2 - 2$. The derivative is $f'(x) = 2x$. An initial guess of 2 is used in each of the approximations below:

```
     ∇ Z←N ITERNEWT X;K
[1]    Z←X                          Initial estimate.
[2]    K←0                          Set iteration counter to 0.
[3]    →4×K<N                       Test if N iterations are done.
[4]    Z←Z-(F Z)÷DF Z               Calculate a new estimate.
[5]    K←K+1                        Increment the iteration counter.
[6]    →3
     ∇
```

In order to define the function F, the name F needs to be available. If a previously defined function F is in the active workspace, it should be erased with the command)ERASE F:

```
     ∇ Z←F X
[1]    Z←¯2+X*2
     ∇

     ∇ Z←DF X
[1]    Z←2×X
     ∇

      0 ITERNEWT 2          1 ITERNEWT 2          2 ITERNEWT 2
2                     1.5                   1.4166666666666665

      3 ITERNEWT 2          4 ITERNEWT 2          5 ITERNEWT 2
1.4142156862745097    1.4142135623746899    1.414213562373095
```

3.11 Statement Separation, Diamond

When several short APL statements are required, it is often convenient to put the statements on the same line. This is effected by placing a diamond ◊ between the statements. The leftmost statement is executed first; for example,

```
      A←5 ◊ B←A*2 ◊ A←10
      A
10
      B
25
      C←?5⍴10 ◊ C ◊ (+/C)÷⍴C
8 2 6 6 9
6.2
```

Although not every modern APL system supports the use of diamond, diamond will be used in this book. The effect of any statement involving diamond can, however, be achieved by writing separate lines.

3.12 *Explicit Output*

Explicit output of results amidst computations can be accomplished using "quad output". This is designated by assigning the desired result to □. For example,

```
      □←3+4
7

      3+□←4
4
7

      (□←5×6)+□←3×4
12
30
42

      S←+/□←?5ρ10
5 8 9 1 4
      S
27

      A←(□←3*2)+□←6
6
9

      A
15

      □←Z←2*5
32

      Z
32
```

The third and fifth examples illustrate a way of explicitly observing the arguments of addition. Explicit output is helpful for debugging and observing intermediate results and gives a simple way for displaying the result of an assignment.

Next consider a modification of the Newton's method function given in Section 3.10. This function combines some of the lines into one using the diamond separator; on line 4, quad output is used to display the iteration count and each intermediate result. Assume that *F* and *DF* are the same as in Section 3.10. Notice that the result of the function is still displayed; it is the last item:

```
      ∇ Z←N ITERNEWT2 X;K
[1]     Z←X ◊ K←0
[2]     →3×K<N
[3]     Z←Z-(F Z)÷DF Z ◊ K←K+1
[4]     □←K,Z
[5]     →2
      ∇

        5 ITERNEWT2 2
1 1.5
2 1.4166666666666665
3 1.4142156862745097
4 1.4142135623746899
5 1.414213562373095
1.414213562373095
```

The branch statements had to be changed since the line numbers changed. ***Remember to check branch statements when lines are inserted or deleted from a function definition.*** Branching to lines by line label avoids this problem. Line labels are treated in Section 10.4.

3.13 Example: Fibonacci Numbers

The Fibonacci numbers are 0 1 1 2 3 5 8 13 21 34 $\cdots$, where terms are the sum of the preceding two terms. That is, the nth Fibonacci number F_n is given by

$$F_0 = 0, \quad F_1 = 1, \quad \text{and} \quad F_{n+1} = F_n + F_{n-1} \quad \text{for } n = 1, 2, 3, \ldots$$

An APL function that computes the nth Fibonacci number uses a simple iteration. Because each term depends on two previous numbers, however, some extra work is required to keep track of the additional information. When line 4 is executed, the iteration counter is incremented and the sense of *OLDZ* and *Z* (current *Z*) is updated:

```
     ∇ Z←FIB N;K;OLDZ;NEWZ
[1]    OLDZ←0 ◊ Z←1 ◊ K←1
[2]    →3×K<N
[3]    NEWZ←Z+OLDZ
[4]    OLDZ←Z ◊ Z←NEWZ ◊ K←K+1
[5]    →2
     ∇
```

```
      FIB 1                FIB 2                FIB 3
1                    1                    2

      FIB 10               FIB 25               FIB 100
55                   75025                3.542248E20
```

Exercises

1. Write an APL function that computes the sample standard deviation, s, of the entries of a vector. In standard mathematical notation,

$$s^2 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n - 1}$$

   where $\bar{x}$ is the average of the entries of the vector $(x_1, x_2, \ldots, x_n)$.
2. Write an APL function *POLYVAL 2* that evaluates a fixed polynomial at several points. The left argument should be a vector representing the polynomial and the right argument should be a vector listing the points at which the polynomial is to be evaluated. See Section 2.6.
3. The distance from the point (x_0, y_0, z_0) to the plane $ax + by + cz + d = 0$ is given by

$$\text{dist} = \frac{|ax_0 + by_0 + cz_0 + d|}{\sqrt{a^2 + b^2 + c^2}}$$

   Write an APL function *DIST* that takes the vector (a, b, c, d) as the left argument, call it *E*, and that takes (x_0, y_0, z_0) as the right argument, call it *P0*, and that results in the distance from the point to the plane.
4. Write an APL function *ANGLETYPE* that has as its arguments two vectors of the same shape and produces a ¯1, 0, or 1, depending on whether the angle between the vectors is obtuse,

right, or acute, respectively. (Hint: Recall the relationship of the dot product to the angle and use the signum function.)

5. Consider Example 3.3, which approximated areas by right-hand endpoints. Using the same *F* as in that example, compare

```
10 RHEP 0 2,        100 RHEP 0 2,        1000 RHEP 0 2,

100 RHEP 2 0,       1000 RHEP ¯2 0,      1000 RHEP 0 ¯2.
```

6. Write an APL function that simulates rolling *N* fair dice.
7. Write an APL function that produces *N* "random" reals in the interval $[a, b]$. The function should have a vector giving the endpoints of the interval $[a, b]$ as the components of its right argument and *N* as its left argument.
8. Experiment with error messages; try the following:

(a)
```
F←100
∇ Z←F W
```

(b)
```
∇ Z←G W
[1]   Z←W*2 ∇
```
followed by `∇Z←A G W` and then by `G←100`

9. Enter the defined functions *PARTITION* and *LHEP*. Notice that *LHEP* invokes a function *F* and the function *PARTITION*:

```
     ∇ Z←N PARTITION AB
[1]    Z←AB[1]+(0,⍳N)×(AB[2]-AB[1])÷N
     ∇

     ∇ Z←N LHEP AB
[1]    Z←((AB[2]-AB[1])÷N)×+/((N⍴1),0)×F N PARTITION AB
     ∇
```

   (a) Describe the result of the functions *PARTITION* and *LHEP* in words.
   (b) Use the function *PARTITION* to rewrite the function *RHEP* given in Section 3.3 in the style of *LHEP* above.
   (c) Write an APL function that approximates integrals using the trapezoidal rule.
   (d) Write an APL function that approximates integrals using Simpson's rule.

10. Rewrite *FAC* given in Section 3.9 using the diamond separator. (Hint: A loop is appropriate.)
11. Write an APL function *THROOT*, with arguments *K* and *A*, which approximates the *K*th root of a given number *A* using 10 steps of Newton's method with initial guess *A*. Write this function so that it uses polynomial evaluation directly rather than using defined functions *F* and *DF*. (Hint: A loop is appropriate.)
12. Use 10 iterations of Newton's method with initial guesses of 2 and 8 to find approximate solutions to $F(x) = 0$, where $F(x) = e^x - x^3 - 3x - 5$. Compute the value of F at the points you find.
13. Write an APL function *POLYDER* that takes a vector giving the coefficients of a polynomial in increasing order as its argument and that results in the vector that represents the derivative of the polynomial. (Hint: The vector that is the same as a vector *V* except that the first entry has been dropped may be constructed by `V[1+⍳¯1+⍴V]`.)
14. Write an APL function *POLYHDER* based on Exercise 13 and a simple loop to give the *N*th derivative of a polynomial of degree at least *N*.
15. Write an APL function that gives the *N*th derivative of a polynomial of degree at least *N* without using a loop.

16. Describe in words the result of the following functions having a nonnegative integer argument *N*:

(a)
```
∇ Z←ITER1 N;K
[1]   Z←0 ◊ K←0
[2]   →3×K<N
[3]   Z←Z+K ◊ K←K+1
[4]   →2
∇
```

(b)
```
∇ Z←ITER2 N;K
[1]   Z←0 ◊ K←0
[2]   →3×K<N
[3]   K←K+1 ◊ Z←Z+K
[4]   →2
∇
```

(c)
```
∇ Z←ITER3 N;K
[1]   Z←⍳0 ◊ K←0
[2]   →3×K<N
[3]   Z←Z,K ◊ K←K+1
[4]   →2
∇
```

(d)
```
∇ Z←ITER4 N;K
[1]   Z←⍳0 ◊ K←1
[2]   →3×K<N
[3]   Z←Z,+/⍳K ◊ K←K+1
[4]   →2
∇
```

(e)
```
∇ Z←ITER5 N;K
[1]   Z←1 ◊ K←0
[2]   →3×K<N
[3]   Z←Z×2 ◊ K←K+1
[4]   →2
∇
```

(f)
```
∇ Z←ITER6 N;K
[1]   Z←1 ◊ K←0
[2]   →3×K<N
[3]   Z←Z×K ◊ K←K+1
[4]   →2
∇
```

17. Write an APL function *ITERSECANT* that estimates roots of a function $f(x)$ by the secant method that is based on

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

where x_n and x_{n-1} are estimates for the root. Let the left argument be *N*, the number of iterations, and the right argument be *X*, a two-element vector giving initial estimates x_0 and x_1. *ITERSECANT* should use a function *F* that defines $f(x)$. (Hint: A loop is appropriate.)

18. Use 15 iterations of the secant method with initial guesses 2 and 8 to estimate roots of $e^x - x^3 - 3x - 5 = 0$. What is the value of *F* at the point you find? See Exercise 17.

19. Modify *ITERNEWT* so that the result of the function is a matrix with the first column giving the iteration number and the second column giving the estimate for the root. (Hints: A loop is appropriate. Initialize the result as a 1-by-2 matrix and catenate a row at each iteration.)

20. Write an APL function *FIBS* that results in a vector of all the Fibonacci numbers up to the *N*th. Use the function to compute the first 50 Fibonacci numbers. (Hints: A loop is appropriate. Initialize the result as the vector 0 1 and catenate successive Fibonacci numbers at each iteration.)

4

Matrix Algebra

This chapter introduces the APL functions for multiplying matrices, solving systems of equations, generating least-squares solutions of systems, computing matrix inverses, and transposing matrices. It presents examples of significant mathematical uses. The chapter also introduces the generalized inner product and reviews the outer product. It uses these operators in the examples and exercises.

4.1 Matrix Product

Multiplication of matrices can be presented as an extension of the dot product of vectors. If the number of columns of matrix *A* equals the number of rows of matrix *B*, the *i, j* entry of the matrix product of *A* and *B* is the dot product of the *i*th row of *A* with the *j*th column of *B*. This gives `+/A[i;]×B[;j]` for the *i, j* entry of the matrix product. This dot product procedure is applied to all possible pairings of rows of *A* with columns of *B* to generate the entries of the matrix product. An operator exists in APL that "applies" the + and × functions to the rows of *A* and columns of *B* and yields the product matrix; it is called the *inner product operator* and is denoted by a dot or period placed between the functions it uses, as in `+.×`.

The inner product operator is used in conjunction with many APL functions with surprisingly useful effects. Its first use was to indicate the conventional matrix product as `A+.×B`. Let

`A←2 2⍴⍳4, B←2 2⍴¯2+⍳4, C←1 2⍴3 6,` and `D←2 1⍴¯1 2.`

Then

```
      A
1 2
3 4

      B
¯1 0
 1 2

      A+.×B
1 4
1 8

      C
3 6

      D
¯1
 2

      C+.×A
21 30

      B+.×D
1
3

      C+.×D
 9

      ρC+.×D
1 1
```

Notice $C+.\times D$ is a 1-by-1 matrix.

The preceding examples are $+.\times$ inner products of matrices with matrices and the results are matrices. This inner product also applies to vectors with matrices and to vectors with vectors. As with the matrix product, there must be a compatible number of components. For example, let $U\leftarrow 3\ 6$ and $V\leftarrow {}^{-}1\ 2$; compare these ***vectors*** to the ***matrices*** C and D:

```
      U+.×A
21 30

      B+.×V
1 3

      U+.×V
9

      ρU+.×A
2

      ρB+.×V
2

      ρU+.×V
(Empty vector.)
```

These shape determinations illustrate that the $+.\times$ product of a vector with a matrix yields a vector, and the $+.\times$ product of two vectors is a scalar—namely, the familiar scalar (dot) product of vector analysis. Thus $U+.\times V$ and $+/U\times V$ have the same result on vectors.

Inner products of matrices with scalars and of vectors with scalars are defined; the scalar is extended to a vector of conforming length. For example, if $V\leftarrow 1\ 2\ 3$ and $A\leftarrow 2\ 3\rho\iota 6$, then $5+.\times V$ and $5+.\times A$ yield the same results as $5\ 5\ 5+.\times V$ and $5\ 5+.\times A$, respectively:

```
      5+.×V
30

      5 5 +.×V
30

      ρ5+.×V
(Empty vector.)

      5+.×A
25 35 45

      5 5 +.×A
25 35 45

      ρ5+.×A
3
```

Examples of inner product functions $f.g$ other than $+.\times$ are in the Exercises and Chapter 5.

4.2 *Example: Markov Processes*

(a) If T is the transition matrix of a Markov process (the columns of T sum to 1), consideration of matrix products of T with itself—that is, powers of T, arise naturally. Using $+.\times$ and assignment

repeatedly, the following expression computes and stores the 2nd, 4th, 8th, 16th, and 32nd powers of the matrix T:

```
      T←3 3 ⍴ .5 .4 0 .3 .4 .2 .2 .2 .8
      T
0.5  0.4  0
0.3  0.4  0.2
0.2  0.2  0.8

      T32←T16+.×T16←T8+.×T8←T4+.×T4←T2+.×T2←T+.×T
      ⎕PP←3
      T2                    T16                   T32
0.37 0.36 0.08    0.222 0.222 0.222    0.222 0.222 0.222
0.31 0.32 0.24    0.278 0.278 0.278    0.278 0.278 0.278
0.32 0.32 0.68    0.5   0.5   0.5      0.5   0.5   0.5
```

Note that all the entries of $T2$ are positive (thus T is regular) and that $T16$ and $T32$ agree when rounded to three significant figures. Of course, a function can be defined for calculating powers of a matrix, but the method used here shows that a simple calculator mode handling of matrix powers is possible.

(b) As is known from the theory of Markov chains, the columns of high powers of T when viewed as vectors will be close to a steady-state vector for the process having transition matrix T. Let V be the vector first column of $T32$ and look at the difference of V and $T+.\times V$:

```
      V←T32[;1]
      V-T+.×V
1.27E¯8     3.18E¯9     ¯1.59E¯8
```

Clearly, $T+.\times V$ is close to V.

4.3 *Solving Linear Systems, Matrix Divide*

The pair of equations

$$x + 2y = 3$$
$$3x + 8y = 5$$

is a system of linear algebraic equations. To solve linear algebraic systems with APL, it is convenient to view them in vector–matrix form. Thus, the system can be written as

$$\begin{bmatrix} 1 & 2 \\ 3 & 8 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

This system is of the form $AX = B$, where A is the coefficient matrix, X is the unknown vector (or column matrix), and B is the right-hand side vector (or column matrix). Any system of m linear algebraic equations in n unknowns can be written as $AX = B$, where A is the m-by-n coefficient matrix of the system and X and B are n-component and m-component vectors or column matrices, respectively. Here square systems ($m = n$) are considered. In Section 4.8, systems with $m > n$ are considered.

Once the matrix A (with independent columns) and the vector B are identified, the solution of the system (if it exists) is readily computed by using a built-in APL function called *matrix divide*, which is denoted by ⌹ and read quad-divide or domino. For example, to solve the system above,

```
      A←2 2 ⍴ 1 2 3 8      Store the coefficient matrix
      B←3 5                and right side vector.
      X←B⌹A                Compute and store the solution.
      X                    Display the solution.
7 ¯2
```

Thus $(x, y) = (7, \overline{\ }2)$ is the solution of the system. The solution procedure can be condensed to

```
      3 5 ⌹ 2 2 ⍴ 1 2 3 8
7 ¯2
```

4.4 Example: Some Solutions of Linear Systems

$$\begin{aligned} 3x - y + z &= \overline{\ }1 \\ 2x \quad + z &= 2 \\ x + y - 3z &= 5 \end{aligned}$$

```
      A←3 3 ⍴ 3 ¯1 1 2 0 1 1 1 ¯3      Store coefficients.
      B←¯1 2 5                         Store right side.
      X←B⌹A                            Solve.
      X
1 4 0
      B-A+.×X                          Check the solution.
0 0 0
```

Since the residue $B-A+.\times X$ is a 0 vector, $(x, y, z) = (1, 4, 0)$ is indeed the solution of the system.

Consider another right-hand side with the same coefficient matrix A:

```
      B←.2 1 .3
      X←B⌹A
      X
0.3125 1.1125 0.375
      B-A+.×X                                  Check the solution.
¯1.110223024E¯16  1.94289029E¯16  4.996003611E¯16
```

Again the residue is 0, within machine precision; hence, X is the solution of the system.

If the columns of A comprise a dependent set of vectors, then $B⌹A$ yields a domain error message. If the columns of A are independent but the system $AX = B$ has no solution in the ordinary sense, $B⌹A$ yields the least-squares solution (see Section 4.8).

Systems having the same coefficient matrix but different right-hand side vectors can be solved simultaneously by forming the right-hand side vectors into a matrix and using domino as before. For example,

$$\begin{bmatrix} 3 & -1 & 1 \\ 2 & 0 & 1 \\ 1 & 1 & -3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -1 & 7 & 1 & 3 & 7 \\ 2 & 9 & 1 & 6 & 1 \\ 5 & 7 & 9 & 5 & 7 \end{bmatrix}$$

is a vector–matrix form of five systems of equations, each having the same coefficient matrix as the system of the preceding example and with right-hand side vectors being the five columns of the matrix on the right. If A is the coefficient matrix and C is the 3-by-5 matrix on the right, then the solutions are given as the columns of $C⌹A$:

```
      C⌹A
1  4   1.5  2.5   2
4  6   1.5  5.5  ¯4
0  1  ¯2    1    ¯3
```

```
      C-A+.×C⌹A
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

All the residues are 0. The first column of $C⌹A$, namely 1 4 0, is the solution obtained above corresponding to the right-hand side vector ¯1 2 5, which is the first column of C.

4.5 *Outer Product*

The outer product operator symbolized by ∘ . and used for generating arrays with elements in prescribed patterns was introduced in Section 2.5. An informal name for this operator is table builder; here it is used to make a times table, a power table, and a 4-by-4 identity matrix:

```
      V←1 2 3 4
      V∘.×V
1 2  3  4
2 4  6  8
3 6  9 12
4 8 12 16
```

```
      V∘.*V
1   1   1   1
2   4   8  16
3   9  27  81
4  16  64 256
```

```
      V∘.=V
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

The outer product will be used to construct Vandermonde matrices in several examples. The Vandermonde matrix associated with a set of numbers is a matrix of nonnegative powers of the numbers. For example, the Vandermonde matrix associated with the numbers ¯2, ¯1, 0, and 4 is

```
      M←¯2 ¯1 0 4∘.* 0 1 2 3
      M
1  ¯2   4  ¯8
1  ¯1   1  ¯1
1   0   0   0
1   4  16  64
```

In general, if a list of numbers is stored as vector X, the Vandermonde matrix for those numbers is given by `X∘.*¯1+⍳⍴X`. Such matrices appear in the interpolation examples below.

At least one APL system has a generalized determinant function, which is denoted with the monadic use of `-.×`. For example,

```
      -.×M
240
```

The nonzero determinant of M illustrates the important fact that the Vandermonde matrix for vectors with distinct entries is nonsingular.

4.6 Example: Polynomial Interpolation

Determine the quartic polynomial that passes through the five points given by the table

t	$^-1$	0	1	2	3
y	0	3	2	9	60

Write the polynomial as $a + bt + ct^2 + dt^3 + et^4 = y$ and find a, b, c, d, and e. Substituting the five t, y pairs in the quartic equation gives a system $AX = B$ of five equations in five unknowns:

$$\begin{aligned}
a - b + c - d + e &= 0 \\
a &= 3 \\
a + b + c + d + e &= 2 \\
a + 2b + 4c + 8d + 16e &= 9 \\
a + 3b + 9c + 27d + 81e &= 60
\end{aligned}$$

The rows of the coefficient matrix A are the 0 through 4 powers of the t values and the components of B are the y coordinates. (Note that A is a Vandermonde matrix.) An outer product will be used to construct A, then `B⌹A` will solve the system:

```
      T←¯1 0 1 2 3
      A←T∘.* 0 1 2 3 4      Construct coefficient matrix.
      B← 0 3 2 9 60         Enter the right-hand side vector.
      B⌹A
3 1 ¯3 0 1
```

The desired quartic polynomial is $3 + t - 3t^2 + t^4$.

4.7 Example: Polynomial Evaluation Revisited

Polynomial evaluation was discussed in Chapter 2, leading to the formula `+/(T*¯1+⍳⍴P)×P`, where P is the vector of coefficients, `¯1+⍴P` is the degree of the polynomial, and T is the number at which the polynomial is to be evaluated. This formula can now be written using the `+.×` matrix product as `(T*¯1+⍳⍴P)+.×P`. This formula has the advantage that if the left argument of `+.×`

is replaced by a matrix, each row of the matrix will be dotted with P just as the single vector $T*{-}1+\iota\rho P$ was dotted with P. With the scalar T replaced by a vector of numbers at each of which the polynomial is to be evaluated, $T\circ .*{-}1+\iota\rho P$ yields a matrix, each row of which is a vector of powers of one of the numbers in T. Then the values of the polynomial at each of the numbers in T are given by $(T\circ .*{-}1+\iota\rho P)+.\times P$.

Evaluating $P(t) = 3 + 2t - 5t^2 + t^4$ at various t values yields

```
      P←3 2 ¯5 0 1          Input the polynomial coefficients.
      T←¯3 1 3 10           Input some arbitrary t values.
      (T∘.*¯1+ι5)+.×P       Compute the polynomial values.
33 1 45 9523
      T←ι10                 Use t values 1 through 10.
      (T∘.*¯1+ι5)+.×P       Evaluate P for these t values.
1 3 45 187 513 1131 2173 3795 6177 9523
      T←2 2ρι4              Try a matrix of t values.
      (T∘.*¯1+ι5)+.×P
 1   3
45 187
```

For frequent use of polynomial evaluation, it is efficient to define a function for using this formula when it is needed. Naming the function *POLYVAL* gives

```
    ∇ Z←P POLYVAL T
[1]   Z←(T∘.*¯1+ιρP)+.×P
    ∇
```

To evaluate $P(t)$ at the t values $^-3$, 1, 3, and 10, just enter

```
      P POLYVAL ¯3 1 3 10
33 1 45 9523
```

Polynomial evaluation is treated again in Sections 9.8 and 9.14.

4.8 *Least-Squares Solutions of Linear Systems*

Many applications lead to linear systems that have more equations than unknowns and may not have solutions in the usual sense. For example, in an experiment to analyze the motion of a falling body, you are led to fitting a quadratic function to many data points. The classical solution of such overdetermined systems using matrix algebra is given in Section 4.12. In this section, the APL primitive that provides an expenditious solution of such systems is presented.

Consider the two systems of four equations in three unknowns:

$$\begin{aligned} x + 2y &= 1 \\ x + z &= 4 \\ y - 3z &= -4 \\ 2x + y + 5z &= 10 \end{aligned} \qquad \begin{aligned} x + 2y &= -5 \\ x + z &= 0 \\ y - 3z &= 3 \\ 2x + y + 5z &= 15 \end{aligned}$$

Both systems have the same coefficient matrix. Store the coefficient matrix and right-hand side vectors in *A*, *B*, and *C*:

```
      A
1  2  0
1  0  1
0  1 ¯3
2  1  5

      B
1  4 ¯4  10

      C
¯5  0  3  15
```

Apply matrix divide with the nonsquare matrix *A* to get least-squares solutions:

```
      B⌹A
3 ¯1  1

      C⌹A
3 ¯1  1
```

Clearly, a check is necessary to determine if $(x, y, z) = (3, -1, 1)$ is a solution of either system. Check by substituting `X←3 ¯1 1` in the expressions `B-A+.×X` and `C-A+.×X`; these residues should be zero vectors if `3 ¯1 1` is a solution:

```
      B-A+.×X
0 0 0 0

      C-A+.×X
¯6 ¯4 7 5
```

Thus, $(x, y, z) = (3, -1, 1)$ is a solution of $AX = B$ but is not a solution of $AX = C$.

With the coefficient matrix A of the preceding example there are infinitely many right-hand side vectors for which the system $AX = B$ has a solution and infinitely many other choices for which it has no solution. In all cases, the expression `X←B⌹A` generates the vector *X* for which `A+.×X` is the closest possible vector to *B*; that is, *X* is the vector for which the magnitude of `B-A+.×X` is least. This magnitude is `0` when the system has a solution. Thus, in the preceding example `X←3 ¯1 1` is the vector for which `A+.×X` is as close to `C←¯5 0 3 15` as possible. For any other choice of *X*, `C-A+.×X` would have greater magnitude than the magnitude of the vector `¯6 ¯4 7 5`.

In linear algebra the vector `A+.×X` is known as the *least-squares approximation* to *B* within the column space of *A* and also as the *projection* of *B* on the column space of *A*. For any linear system $AX = B$ in which the columns of *A* are linearly independent, `B⌹A` yields the least-squares solution to the system; in particular, when the system has a solution in the usual sense, `B⌹A` gives the solution.

4.9 Example: Least-Squares Polynomial Curve Fitting

Determine the best (in the least-squares sense) cubic polynomial fit for the data points

t	0	1	2	3	4	5
y	0	.2	.35	.45	.5	.55

Find a, b, c, and d so that $a + bt + ct^2 + dt^3$ best fits the data. Substituting the six t values in this cubic expression and equating to the y values gives the following system $AX = B$ of six equations:

$$
\begin{aligned}
a &= 0 \\
a + b + c + d &= .2 \\
a + 2b + 8c + 16d &= .35 \\
a + 3b + 9c + 27d &= .45
\end{aligned}
$$

$$a + 4b + 16c + 64d = .5$$

$$a + 5b + 25c + 125d = .55$$

The coefficient matrix is a Vandermonde matrix and has independent columns. Although it is unlikely for this system to have a solution, `B⌹A` will yield the coefficients of the least-squares approximation:

```
      ⎕PP←3
      T←0,⍳5
      A←T∘.*0 1 2 3                    Construct coefficient matrix.
      B←0 .2 .35 .45 .5 .55
      X←B⌹A                            Compute least-squares solution.
      X
¯0.00159  0.242  ¯0.0379  0.00231
```

To three significant figures the least-squares cubic polynomial is $-.00159 + .242t - .0379t^2 + .00231t^3$.

Should you want the least-squares quadratic approximation, the only modification of the APL formulas in the preceding solution is to drop the 3 from the expression `A←T∘.*0 1 2 3`. Similar minor changes in the right argument of the `∘.*` function for constructing the coefficient matrix of the system would yield the coefficients of the least-squares polynomial with degree (up to 5) of your choice.

4.10 Matrix Inverse

Dyadic uses of quad-divide, ⌹, were discussed in Sections 4.3 and 4.8; quad-divide can also be used monadically. If `A` is a nonsingular matrix, `⌹A` yields the *matrix inverse* of `A`. For example, let `A←3 3⍴1 5 4 ¯1 2 1 0 3 2`, then

```
      A                ⌹A                A+.×⌹A
 1  5  4          ¯1 ¯2  3          1  0  0
¯1  2  1          ¯2 ¯2  5          0  1  0
 0  3  2           3  3 ¯7          0  0  1
```

For a nonsingular matrix `A`, both `A+.×⌹A` and `(⌹A)+.×A` yield the identity matrix of the same size as `A`. Moreover, for a linear system $AX = B$ with nonsingular coefficient matrix A, an alternate formula for the solution vector is `X←(⌹A)+.×B`.

If `A` is a nonsquare matrix with independent columns, `⌹A` yields a left inverse of `A`. If the columns of `A` are dependent, a domain error message results from `⌹A`.

4.11 Matrix Transpose

Many matrix algebra computations involve transposes of matrices. APL has a *transpose* function, denoted by `⍉`, which interchanges rows and columns in the conventional way. For example,

```
      A          ⍉A          ⍉⍉A
1 2 3          1 4          1 2 3
4 5 6          2 5          4 5 6
               3 6
```

4.12 Example: A Formula Using Transpose

If the columns of a coefficient matrix A are linearly independent, a formula for the least-squares solution of the linear algebraic system $AX = B$ is $X = (A'A)^{-1}A'B$, where A' signifies the transpose of A. This formula translates directly to the APL expression `X←(⌹(⍉A)+.×A)+.×(⍉A)+.×B`, which is a "program" for computing `X`, given `A` and `B`. Of course, the dyadic use of domino, `X←B⌹A`, is an easier, more efficient way to compute the least-squares solution using APL.

4.13 Example: Gram–Schmidt Orthogonalization

The Gram–Schmidt process is used to replace a set of linearly independent vectors by a set of orthonormal vectors spanning the same space. Two defined functions will make application of the process more transparent:

```
     ∇ Z←UNIT V
[1]    Z←V÷(V+.×V)*.5          Divide V by its length; "unitize" V.
     ∇

     ∇ Z← U PROJ V
[1]    Z←V×(U+.×V)÷V+.×V       Projection of U onto V.
     ∇
```

If `V1`, `V2`, and `V3` are linearly independent n vectors, the Gram–Schmidt process produces orthonormal vectors `U1`, `U2`, and `U3` via

```
U1←UNIT V1
U2←UNIT V2-V2 PROJ U1
U3←UNIT V3-(V3 PROJ U2)+V3 PROJ U1
```

Look at the steps in generating `U2`, for example: first `V2` is projected on `U1`, the result is then subtracted from `V2` yielding the complementary projection, finally that result is normalized and assigned to `U2`.

Applying these formulas with `V1←1 2 2 ¯2 1`, `V2←2 2 3 6 1`, and `V3←0 2 6 ¯1 0` produces

```
      U1
0.267 0.535 0.535 ¯0.535 0.267
      U2
0.263 0.253 0.389 0.836 0.126
      U3
¯0.487 ¯0.29 0.712 ¯0.0281 ¯0.414
```

4.14 Example: Testing for Orthonormality

The column vectors of a matrix A are an orthonormal set of vectors if and only if $A'A$ is an identity matrix. The set of vectors `U1`, `U2`, and `U3` generated by the Gram–Schmidt process in Example 4.13

will be tested by this criterion. Suppose *U1*, *U2*, and *U3* are already in the active workspace:

```
      A←⍉3 5⍴U1,U2,U3          Form matrix with U's as columns.
      (⍉A)+.×A                 Apply the test.
1.00E0    2.78E¯17  1.67E¯16
2.78E¯17  1.00E0    2.98E¯17
1.67E¯16  2.78E¯17  1.00E0
```

Within computer capability this is a 3-by-3 identity matrix, thereby confirming the orthonormality of the set of *U*'s. Observe that in forming *A* it was convenient first to make a matrix having the *U*'s as row vectors and then to transpose that to get *A*.

4.15 The Power Method

Eigenvalues and eigenvectors play an important role in analyzing the behavior of processes described with matrices. An eigenvalue λ of a square matrix A is a scalar such that $AU = \lambda U$ for some nonzero vector U. Such a vector U is called an eigenvector of A associated with λ. It is usually difficult to attack the problem of finding the eigenvalues of a matrix by directly computing the characteristic equation and locating its roots. This section provides an introduction to estimating eigenvectors and eigenvalues. The power method is a simple method for estimating an eigenvector associated with a dominant eigenvalue; that is, an eigenvalue that is larger in magnitude than the other eigenvalues of the matrix. Other easy algorithms for finding eigenvalues and computing error bounds for those estimates are found in the exercises.

The power method is a simple iteration beginning with an initial estimate X_0 for the eigenvector associated with the dominant eigenvalue. An estimate X_k for the eigenvector is multiplied by the matrix A and the result is scaled to unit length:

$$Y_{k+1} = AX_k$$

$$X_{k+1} = \frac{Y_{k+1}}{\|Y_{k+1}\|}$$

At the kth iteration the vector computed is a unit vector in the A^kX_0 direction.

The power method works since when X_0 is written in terms of a basis of eigenvectors, the contribution to A^kX_0 corresponding to the eigenvector associated with the dominant eigenvalue becomes the largest term. At each step the estimated eigenvector is scaled to unit length to prevent overflow, since the components of A^kX_0 tend to get very large as k increases.

Using the function *UNIT* from Section 4.13, the two main steps of the power method can be written as *X←UNIT A+.×X*, where *X* is the current estimate for the eigenvector. The function *POWMETH* below iterates these steps *N* times and results in a matrix. On line 1, the output variable *Z* is initialized to be a matrix with one row that is the unit vector in the direction of the initial estimate of the eigenvector. On line 3, the estimate of the eigenvector at the *k*th step is computed then catenated to the output variable *Z*. The right argument of *POWMETH* is an initial estimate of an eigenvector of *A*. Notice that on line 3 the function uses the global variable *A*, which is the matrix for which the eigenvector associated with a dominant eigenvalue is being computed. In the example, ten iterations are

computed using as an initial guess the vector 1 1 1:

```
     ▽ Z←N POWMETH X;K
[1]    K←0 ◊ Z←(1,ρX)ρX←UNIT X
[2]    →3×K<N
[3]    Z←Z,[1]X←UNIT A+.×X ◊ K←K+1
[4]    →2
     ▽

       A
6 1 1
2 4 1
0 1 2
       T←10 POWMETH 1 1 1
       T
0.57735 0.57735 0.57735
0.72429 0.63375 0.27161
0.76541 0.62025 0.17156
0.7819  0.6075  0.1399
0.78951 0.60006 0.1288
0.79318 0.59614 0.12446
0.79497 0.59414 0.1226
0.79584 0.59314 0.12175
0.79627 0.59264 0.12135
0.79649 0.59239 0.12115
0.79659 0.59227 0.12106
```

Notice that the estimates for the eigenvector appear to be converging. Some natural questions to ask are: How quickly is it converging? How important is the choice of the initial eigenvector? How do you get an estimate of the eigenvalue once the eigenvector has been estimated? You can refer to numerical analysis texts for a complete discussion of these questions.

The question of how to estimate the eigenvalue once an eigenvector is estimated has a satisfying answer. First, notice that if A has eigenvector λ associated with eigenvector U, then $AU = \lambda U$ so $AU - \lambda U$ is the 0 vector; thus, $\|AU - \lambda U\| = 0$. When an estimated eigenvalue μ and eigenvector X have been computed, it is natural to look at the *residue* $AX - \mu X$. The estimates are good if and only if the length of the residue $\|AX - \mu X\|$ is near zero. If only an estimate for the eigenvector is known, then the best estimate for the eigenvalue can be computed by the Rayleigh quotient given by the theorem:

If A is a real matrix and X is a nonzero vector, then the estimate for eigenvalue μ that minimizes the length of the residue $\|AX - \mu X\|$ is given by the *Rayleigh quotient*

$$\mu = \frac{X \cdot (AX)}{X \cdot X}$$

If X is a unit vector, then the denominator is 1. In that case the Rayleigh quotient can be expressed by `X+.×A+.×X` in APL. For the example considered above, the best estimate for eigen-

vector, X, is given by the last row of T. Next, the Rayleigh quotient M is computed and the residue RES is examined:

```
      X←T[11;]
      X
0.79659 0.59227 0.12106
      M←X+.×A+.×X
      M
6.8950
      M×X
5.4925 4.0837 0.83471
      A+.×X
5.4929 4.0833 0.83439
      RES←(A+.×X)-M×X
      RES
0.00035486 ¯0.00041223 ¯0.00031823
```

Notice that RES suggests M and X are estimates that are good to about three significant figures.

Exercises

1. Use the function *POLYVAL* of Example 4.7 and the function *PARTITION* in Exercise 3.9 to write a brief expression that yields the values of $4 - t + 2t^2 - t^3$ at points uniformly spaced one-tenth of a unit apart along the interval from ¯1 to 2.
2. **(a)** Find a steady-state vector (to three significant figures) for the transition matrix
   `T←3 3⍴0 0 .1 .2 .7 .3 .8 .3 .6.`
   (b) Not every transition matrix has a steady-state vector. Compare the even and odd powers of
   `T←3 3⍴0 .25 0 1 0 1 0 .75 0.`
3. Write a monadic function *VANDERMONDE* that generates the Vandermonde matrix corresponding to a given vector X.
4. **(a)** Write the formula for *U4* that would be next in the list of formulas for the Gram–Schmidt process given in Example 4.13.
   (b) Apply the Gram–Schmidt process to the set of four independent 5-vectors:
   `V1←0 ¯1 0 6 ¯1, V2←6 1 7 6 ¯3, V3←4 7 0 5 0,`
   `V4←4 ¯3 ¯2 ¯2 5.`
   (c) Check the orthonormality of your solution to part (b) using the test described in Example 4.14.
5. Use the function `∘.=` to construct a 10-by-10 identity matrix.
6. Solve this system by two methods:

$$\begin{aligned} 2x - y + z &= 12 \\ 5x + y + z &= 12 \\ 4x + 3y - 2z &= {}^{-}11 \end{aligned}$$

   (a) Use ⌹ monadically to find the inverse of the coefficient matrix.
   (b) Use ⌹ dyadically.

7. Determine the cubic interpolating polynomial for the four points given by this table:

t	-2	-1	1	2
y	-23	4	-2	13

8. Find the (a) linear, (b) quadratic, (c) cubic, and (d) quartic least-square polynomial approximations to these data points:

t	0	1	2	3	4
y	-4	0	72	368	1140

9. For equal length vectors U and V, the inner product `U+.×V` gives the *sum* of the *products* of corresponding entries in the vectors.
   (a) Express in words what the value of `U+.=V` tells about the vectors.
   (b) Experiment with other dyadic scalar functions f and g to form other inner products $f.g$. Using simple vectors, try `U+.*V`, `U×.*V`, `U×.-V`, and so on. Also use scalars; for example, `5+.=V` and `5×.-V`.
   (c) Let `B←⍳13` and `U←13⍴1 0`; describe in words the results of `B+.×U` and `B×.*U` (Compare your answer to that of Exercise 1.6).
   (d) Let `A←⍳100` and `V←100⍴0 1 0 0 1`; describe in words the results of `A+.×V` and `A×.*U`.
10. Determine the APL evaluation of each expression; check your answers with APL. Let `A←-⍳4` and `B←2 4⍴⍳8`.

    (a) `A` (b) `⌹A`
    (c) `⌹1 4⍴A` (d) `⌹4 1⍴A`
    (e) `B` (f) `⌹B`
    (g) `⌹⌹B` (h) `B,[1]A`
    (i) `⌹B,[1]A`

11. Generate two "random" matrices A and B using `A←?3 2⍴10` and `B←?2 3⍴10`. Display A and B and compare `⌹A+.×B` with `(⌹B)+.×⌹A`. Also compare `⌹B+.×A` with `(⌹A)+.×⌹B`.
12. Let `C←2 3 4⍴⍳24` and `D←4 3 2⍴⍳24`. Compute and display

    (a) `C` (b) `⌹C`
    (c) `⌹⌹C` (d) `D`
    (e) `⌹D` (f) `⌹⌹D`
    (g) `⍴⌹C` (h) `⍴⌹D`

13. Use *POWMETH* of Section 4.15 on the matrix `A1` with initial estimate `1 1 1` for the eigenvector. Compute ten iterations.

```
      A1
2 4 4
3 8 4
3 3 2
```

    (a) Give the last estimate for the eigenvector.
    (b) Give the best estimate for the eigenvalue.
    (c) Compute the residue.

14. One disadvantage of the power method is that it only determines eigenvectors of dominant eigenvalues. Suppose A^{-1} exists and A has eigenvalue λ associated with eigenvector U. It is easy to check that A^{-1} has eigenvalue $1/\lambda$ associated with the same vector U. Thus, if A^{-1} has a domi-

nant eigenvalue, the power method with suitable initial guess applied to A^{-1} converges to an eigenvector associated with the dominant eigenvalue of A^{-1}, and the same vector is associated with the smallest magnitude eigenvalue of A. This algorithm is called the *inverse power method.* Use the inverse power method on the matrix ***A1*** from Exercise 13 with initial estimate 1 1 1 for the eigenvector. Compute ten iterations. (Hint: Use ***POWMETH*** with an appropriate choice of matrix.)

(a) Give the last estimate for the eigenvector.
(b) Give the best estimate for the smallest magnitude eigenvalue of A.
(c) Compute the residue.
(d) Repeat (a)–(c) with 50 iterations.

15. One disadvantage of the power and inverse power methods is that they only determine eigenvectors associated with smallest nonzero or largest magnitude eigenvalues. Consider the *shifted inverse power method.* Suppose a matrix A has eigenvalue λ associated with eigenvector U and $(A - \alpha I)^{-1}$ exists, where I is an identity matrix and α is a scalar "shift." It is easy to check that $(A - \alpha I)^{-1}$ has an eigenvalue $1/(\lambda - \alpha)$ associated with the same vector U. Notice that $1/(\lambda - \alpha)$ is large in magnitude if and only if $\lambda - \alpha$ is small in magnitude. Thus, if $(A - \alpha I)^{-1}$ has a dominant eigenvalue, the power method with suitable initial guess applied to $(A - \alpha I)^{-1}$ converges to an eigenvector associated with the dominant eigenvalue of $(A - \alpha I)^{-1}$, and the same vector is associated with the eigenvalue of A nearest to α. Use the shifted inverse power method with an initial estimate 1 1 1 and a shift of $\alpha = 3$ to estimate the eigenvalue nearest to 3 for the matrix ***A1*** from Exercise 13. Compute ten iterations. (Hint: Use ***POWMETH*** with an appropriate choice of matrix.)

(a) Give the last estimate for the eigenvector.
(b) Give the best estimate for the eigenvalue nearest to 3.
(c) Compute the residue.
(d) Repeat (a)–(c) with $\alpha = -1$.

16. The *Rayleigh quotient iteration* is based upon the shifted inverse power method, with the shift updated at each iteration to be the estimated eigenvalue given by the Rayleigh quotient. Since each inverse matrix is used for only one step, it is more efficient to solve a system than to multiply an estimated eigenvector by the inverse of a matrix. Thus, the step of the shifted inverse power method $Y_{n+1} = (A - \mu_n I)^{-1} X_n$ is replaced by the following: Find the solution Y_{n+1} to the system $(A - \mu_n I)Y_{n+1} = X_n$. The Rayleigh quotient iteration begins with an estimate for eigenvector X_0 and eigenvalue μ_0. Then the estimated eigenvector and eigenvalue are updated as follows:

Y_{k+1} is the solution to $(A - \mu_k I)Y_{k+1} = X_k$.
X_{k+1} is the unit vector in the direction of Y_{k+1}.
μ_{k+1} is computed by the Rayleigh Quotient on X_{k+1}.

Use the Rayleigh quotient iteration with initial guess 1 1 1 for the eigenvector and initial estimate 3 for the eigenvalue for the matrix ***A1*** from Exercise 13. Compute three iterations.

(a) Give the last estimate for the eigenvector.
(b) Give the best estimate for the eigenvalue.
(c) Compute the residue.

17. It is easy to test how good an estimate for an eigenvalue is when the matrix is ***symmetric.*** Let μ be an estimate for an eigenvalue of a symmetric matrix A associated with an estimated eigenvector X of unit length. Let λ denote an actual eigenvalue of A nearest to μ. The length of the residue gives an error bound on the estimate for the eigenvalue $|\lambda - \mu| \leq \|AX - \mu X\|$.

A2

1 2 3
2 5 4
3 4 2

(continued)

For the matrix *A2*, estimate eigenvectors several ways. In each case use 1 1 1 as an initial guess for an eigenvector and give the following: the last estimate for the eigenvector, the best estimate for the eigenvalue, the residue, and a bound for the error in the estimate for the eigenvalue.

(a) Use ten iterations of the power method.

(b) Use ten iterations of the inverse power method.

(c) Use ten iterations of the shifted inverse power method with a shift of $\alpha = 2$.

(d) Use five iterations of the Rayleigh quotient iteration with initial estimate for the eigenvalue of 2.

18. Write a function *GRAMSCH* that applies the Gram–Schmidt process to the columns of a matrix with independent columns given as its right argument. The result should be the matrix with orthonormal columns generated by the Gram–Schmidt process. Test your function on the matrix with columns that are the vectors in Example 4.13. (Hints: A loop is appropriate. Use the fact that the projection of column *K* of a matrix *A* onto the column space of the preceding *K*−1 columns is given by the expression *A*[;⍳*K*−1]+.×*A*[;*K*]⌹*A*[;⍳*K*−1].)

19. A simple form of the QR algorithm for diagonalizing a square matrix A_0 is given by iterating the following:

Q_k is the orthogonal matrix produced by applying *GRAMSCH* to A_k.
An updated matrix $A_{k+1} = Q_k' A_k Q_k$ has the same eigenvalues as A_k.

Write a function to apply *N* iterations of the QR algorithm to a nonsingular matrix. Apply the function with 5, 10, 15, and 20 iterations to the matrix *A1* in Exercise 13. (Hint: A loop is appropriate.)

20. Redo Exercise 19 with the matrix *A2* from Exercise 17.

5

Data Comparison and Logical Functions

This chapter introduces a host of APL functions useful for solving a wide variety of problems as well as for building more complicated functions. These functions are used in what is traditionally called programming — more on that in Chapter 10. It also illustrates solution of some basic data manipulation problems.

The functions introduced are the relationals or comparatives, the logicals, floor, ceiling, and membership. Many of them are incorporated in generalized inner products introduced here. The chapter closes with a discussion of the system variable comparison tolerance.

5.1 Relational Functions: Comparatives

The APL relational functions, or comparatives, are denoted by the dyadic use of the conventional symbols <, ≤, =, ≥, >, and ≠. The relational functions test the relative values of expressions; they yield results of 1 if the indicated relation is true and 0 if it is false. These results can be used as arguments for other functions. For example,

```
      2>3
0

      2≤3
1

      2≠3
1

      1+2>3
1

      1+2≤3
2

      3-1≠3
2
```

The relationals are scalar functions; thus, numeric arrays serve as arguments in the usual ways; for example,

```
      4 1 3≠3 2 3
1 1 0

      2=5 2 1 2 3
0 1 0 1 0

      2<2 3 ⍴⍳6
0 0 1
1 1 1
```

The relational functions handle many tabulating problems easily. For example, let V be a vector,

```
      V
5 2 0 3 3 1 9 6 8 4 3 2 8 9 3 1 1 1 3

      3=V                                  Locate 3s in V.
0 0 0 1 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1

      +/3=V                                Count 3s in V.
5

      +/4>V                                Count entries smaller than 4.
12
```

One way to count the number of entries between 3 and 6 inclusive is to subtract the number of entries less than 3 from the number less than or equal to 6 as follows:

```
      (+/V≤6)-+/V<3
8
```

Using an outer product, counts on several different numbers can be made simultaneously:

```
      3 2∘.=V                              Locations of 3s and 2s in V.
0 0 0 1 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1
0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

      +/3 2∘.=V                            Counts of 3s and 2s in V.
5 2

      +/(0,⍳9)∘.=V                         Counts of each of the digits in V.
1 4 2 5 1 1 1 0 2 2

      +/(0,⍳9)∘.≥V                         Counts of entries less than or
1 5 7 12 13 14 15 15 17 19                 equal 0,1,···,9, respectively.
```

5.2 *Example: Averages on Selected Scores*

Let the nonnegative entries of S contain the scores of seven students on eleven 10-point quizzes. A ¯1 entry indicates an excused absence from a quiz. The example is

```
      S
 7  7  6  8  7  8  6  9  9  8 10
¯1  6  8  9  8 10 10  8  6  8 10
 8  9 10  9 10 10  9 10  8 10 10
 6  7  5  5  7  7  8  6  7 ¯1  8
 5  9  6  8  8  6  5  8 10  4  8
¯1 10 10  9  9 10 ¯1 10  9  8 10
 8  8 10 ¯1 ¯1  7  9 10 ¯1  7 10
```

Find the average quiz score for each student, not counting quizzes from which the student was excused. Set `⎕PP←2`; the required averages are then given by

```
      (+/S×S≥0)÷+/S≥0
7.7  8.3  9.4  6.6  7  9.4  8.6
```

5.3 *Logical Functions*

You can accomplish calculations using the basic functions of mathematical logic with the APL functions *not* `~`, *and* `∧`, *or* `∨`, *nand* `⍲`, and *nor* `⍱`. The domains of these functions are restricted to arrays of 0s and 1s, and their results are also 0s and 1s. Variables comprised of only 0s and 1s are termed *Boolean variables*.

The logical negation function, *not*, is monadic and reverses the logical parity of its argument; thus,

```
      ~0          ~~0          ~0 0 1 0 1
1                 0            1 1 0 1 0
```

The definitions of the dyadic logical functions are summarized by their truth tables as follows:

and: `∧`

	0	1
0	0	0
1	0	1

or: `∨`

	0	1
0	0	1
1	1	1

nand: `⍲`

	0	1
0	1	1
1	1	0

nor: `⍱`

	0	1
0	1	0
1	0	0

Thus,

```
      1∧1          1∨0          1⍲1
1                 1            0
```

The truth tables above can be generated using outer products as follows:

```
      0 1∘.∧0 1          0 1∘.∨0 1
0 0                 0 1
0 1                 1 1
```

The tabulating capabilities illustrated in Section 5.2 can be extended with the use of the logical functions. Again, let *V* be the vector from the last section:

```
      V
5 2 0 3 3 1 9 6 8 4 3 2 8 9 3 1 1 1 3

      (2=V)∨3=V          Show locations of 2s and 3s in V.
0 1 0 1 1 0 0 0 0 0 1 1 0 0 1 0 0 0 1

      +/(2=V)∨3=V        Count the 2s and 3s in V.
7

      +/(3≤V)∧V≤6        Count entries between 3 and 6 inclusive.
8
```

```
      +/(V<3)∨6<V      Count entries less than 3 or greater than 6.
11

      +/V×V>3          Sum entries greater than 3.
49
```

5.4 *Example: Frequency Distribution*

To construct a frequency distribution of a data set, you will want to count the data points that lie within each of several specified intervals. For example, the number of digits of the vector V (from Section 5.1) that lie in each of the intervals [0, 3], (3, 6] and (6, 9] is given by

```
      +/(¯1 3 6∘.<V)∧3 6 9∘.≥V
12 3 4
```

Observe that each of the expressions `0 3 6∘.<V` and `3 6 9∘.≥V` generates a three-row matrix showing the locations of entries of V that satisfy the respective indicated relation. Then the logical and, ∧, of these matrices shows where both relations are satisfied; for example, the third row will yield a `1` in each column that corresponds to the position of a `7`, `8`, or `9` in V. Finally, the row sums, `+/`, of this matrix give the desired counts.

5.5 *Generalized Inner Product: Vectors*

The APL formula `+/U×V` for the ordinary inner (dot) product of vectors U and V of matching length was given in Section 1.12. This inner product is the sum of products of corresponding entries of U and V. That scheme for combining two functions is useful for functions other than + and ×. For example, the sum of equals of corresponding entries, `+/U=V`, gives the number of positions in which U and V have the same entry.

The APL *inner product* operator combines the effects of two functions in the manner shown in these examples; the operator is denoted by a dot or period placed between its function arguments, as in `+.×` and `+.=`. In general if f and g are any scalar dyadic functions, the inner product function $f.g$ when applied to equal length vectors U and V yields the f reduction of the vector resulting from applying g to corresponding entries of U and V. Thus, for vectors U and V, the formulas $Uf.gV$ and f/UgV yield exactly the same results. However, $f.g$ inner product functions apply to arrays X and Y other than vectors, and for such arrays the notation f/XgY might be meaningless or yield a result different from $Xf.gY$.

Here are some inner product functions and their effects when applied to equal length vectors:

`U+.*V`	Sums V powers of entries of U.
`U×.*V`	Multiplies V powers of U entries; if V is Boolean, it selects entries of U and multiplies them.
`U∨.=V`	Tests whether U and V agree in at least one position.
`U∧.=V`	Tests whether U and V agree in every position.

`U∧.≠V` Tests whether *U* and *V* differ in every position.

`U∧.>V` Tests whether each entry of *U* is greater than the corresponding entry of *V*.

There are many more such examples.

Here are a few examples with specific vectors:

```
      U←8 0 3 1 9 2 5 4 6 1 9
      V←0 8 4 1 2 5 9 4 5 9 1

      U+.×V                    UV.=V                    U+.=V
150                        1                        2
```

5.6 Example: Weighted Average

Weighted averages of sets of numbers have various uses. The weighted average of a list of numbers $x_1, x_2, \cdots, x_n$ with corresponding nonnegative weights $w_1, w_2, \cdots, w_n$ is defined as

$$(x_1w_1 + x_2w_2 + \cdots + x_nw_n) \div (w_1 + w_2 + \cdots + w_n)$$

If *X* is a vector of numbers and *W* is a vector of weights, the weighted average is given by the APL expression `(X+.×W)÷+/W`. For example;

```
      X←0 1 8 ◊ W←1 4 1 ◊ (X+.×W)÷+/W
2
```

5.7 Generalized Inner Product: General Arrays

If *A* and *B* are matrices, *Af*.*gB* inner products are defined if the number of columns of *A* equals the number of rows of *B*. When that conformability condition is met, the result of *Af*.*gB* is a matrix with the *i*th row, *j*th column element being the *f*.*g* inner product of the *i*th row vector of *A* and the *j*th column vector of *B*; that is, *A*[*i*;]*f*.*gB*[;*j*]. Thus, the inner product of matrices is formed from the inner products of row vectors of *A* with column vectors of *B*. In particular, the +.× inner product gives the conventional matrix product of *A* and *B*. For example,

```
      A                    B                    A+.×B
1 ¯1                 3 2  1               3 1 2
2  1                 0 1 ¯1               6 5 1

      A∧.=B                A+.×A                B+.×B
0 0 1                ¯1 ¯2                (yields error message)
0 1 0                 4 ¯1
```

Inner products of matrices with vectors are defined if the matrix and vector conform: If the length of vector *U* matches the number of rows of matrix *A*, then *Uf*.*gA* is defined. If the number of columns of matrix *A* matches the length of vector *V*, then *Af*.*gV* is defined. The result in each case

is a ***vector.*** Letting $U \leftarrow 1\ \ {}^{-}1$ and $V \leftarrow 1\ \ 2\ \ {}^{-}2$ and using the matrix B above gives

```
      U+.×B                    ρU+.×B
3 1 2                          3

      B+.×V                    ρB+.×V
5 4                            2
```

The shapes of $U+.\times B$ and $B+.\times V$ being single numbers confirms that these are vectors. Note that $B+.\times V$ can be interpreted as the V linear combination (sum of multiples) of the column vectors of B.

Compare the results of using row and column matrices in place of the vectors in these calculations:

```
      UMAT←1 2ρU               UMAT+.×B                 ρUMAT+.×B
                         3 1 2                    1 3

      VMAT←3 1ρV               B+.×VMAT                 ρB+.×VMAT
                           5                      2 1
                           4
```

The inner products of B with $UMAT$ and $VMAT$ yield ***matrices.***

The inner product $\wedge.=$ is especially handy. If the matrix A conforms with the vector U, then $U\wedge.=A$ yields a Boolean vector with a 1 indicating a column of A that matches U in every position, and, similarly, for row matches of A with V using $A\wedge.=V$.

Let the vectors U, V, and the 5-by-12 matrix A be as shown:

```
      U
5 3 5 4 2
      V
1 4 2 4 1 2 4 5 2 1 3 1
      A
1 5 4 1 6 1 4 1 3 5 5 1
1 3 2 3 1 4 2 6 3 3 3 5
4 5 4 5 2 6 4 5 5 6 5 1
1 4 2 4 1 3 4 5 2 1 4 1
4 2 5 4 2 1 3 4 1 3 2 3

      U∧.=A                    Find columns of A that match U.
0 1 0 0 0 0 0 0 0 0 1 0

      +/U∧.=A                  Count of columns that match U.
2

      A∧.=V                    Find rows of A that match V.
0 0 0 0 0

      1 1 0 0 1+.×A            Sum corresponding entries of
6 10 11 8 9 6 9 11 7 11 10 9   rows 1, 2, and 5 of A.

      A+.×12ρ1 0               Sum corresponding entries of
23 12 24 14 17                 the odd numbered columns of A.
```

Inner products of scalars with arbitrary arrays are defined; the scalar is automatically extended to a vector of conforming length. Thus, using the matrix *A* above and the vector *W*←1 2 3, the expressions 5*f*.*gA* and *Wf*.*g*5 are treated as 5 5 5 5 5*f*.*gA* and *Wf*.*g*5 5 5, respectively. For example,

```
      5∧.≥A                                  W+.×5
1 1 1 1 0 0 1 0 1 0 1 1          30
```

Note that the inner product 5∧.≥*A* finds the columns of *A* with no entry larger than 5.

Consider nonscalar arrays *A* and *B* of arbitrary dimension and dyadic scalar functions *f* and *g*. The inner product *Af*.*gB* is defined if the last axis of *A* has the same length as the first axis of *B*, and, then, the result has dimension 2 less than the sum of the dimensions of *A* and *B*. The shape of *Af*.*gB* is the shape of *A* less its last component catenated with the shape of *B* less its first component; for example, if *A* is a three-dimensional array of shape 3 2 4 and *B* is a matrix of shape 4 2, then *Af*.*gB* has dimension 3 and has shape 3 2 2.

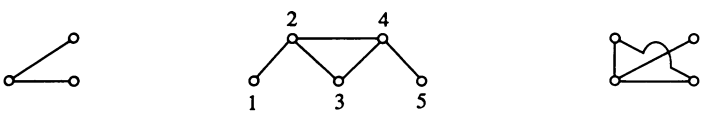
5.8 *Example: Paths in Graphs*

A finite collection of vertices in a plane together with some edges joining pairs of distinct vertices is called a graph. In the three small graphs in Figure 5.1, each edge provides a *path* from one vertex to another. Also, an ordered set of n edges with successive pairs concurrent end to end is a *path of length* n. For example, in Figure 5.1(b), the succession of edges from vertex 1 to 2, then 2 to 4, and then 4 to 3 is a path of length 3 from vertex 1 to 3. A graph with numbered vertices has an associated *adjacency* matrix; this is a Boolean matrix with a 1 in the (i, j) position if and only if there is an edge joining vertices i and j.

The adjacency matrix of Figure 5.1(b) is

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The adjacency matrix marks with a 1 the pairs of vertices having a path of length 1 joining them. This suggests a ***problem:*** From the adjacency matrix A of a graph having n numbered vertices, find the matrix that marks with a 1 the pairs of vertices having a path of length 2 joining them.



(a) (b) (c)

Figure 5.1 Three small graphs.

For Figure 5.1(b), the matrix required is easily found directly from the graph to be

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

In general to have a path of length 2 from one vertex to another, say i to j, there must be a path of length 1 (an edge) from i to some intermediate vertex, say k, and a path of length 1 from k to j. If for *any* $k = 1, 2, \cdots, n$, both the (i, k) *and* (k, j) entries of A are 1s, then there is a path of length 2 and the (i, j) entry of the required matrix is a 1. For example, to use the matrix A of Figure 5.1(b) to test for a path of length 2 from vertex 3 to 4, determine whether any pairs of corresponding entries in row 3 and column 4 are both 1s:

row 3: 0 1 0 1 0

column 4: 0 1 1 0 1

In fact, the second entries in row 3 and column 4 are 1s, and thus there is a path of length 2 from 3 to 4 via vertex 2. The procedure then is to evaluate the logical or, ∨, of the logical ands, ∧, of corresponding entries in row i and column j of A for each i and j from 1 to n. Thus, the required matrix is given in APL as the inner product `A∨.∧A`.

By extending this argument it can be shown that the Boolean matrix `A∨.∧A∨.∧A` has a 1 in the (i, j) position if and only if there is a path of length 3 from vertex i to j, and similarly for longer paths. See Exercise 15.

5.9 *Floor, Ceiling*

If X is any real number, the *floor* of X, denoted `⌊X`, yields the largest integer that is not larger than X. This is also known as the greatest integer function. Thus,

```
      ⌊2.3
2
      ⌊1.9
1
      ⌊2 ¯1.5 .4 ¯.3
2 ¯2 0 ¯1
```

The formula `⌊.5+X` rounds X to the nearest integer, with numbers having fractional part .5 rounded to the next larger integer:

```
      ⌊.5+2.3
2
      ⌊.5+1.9
2
      ⌊.5+2 ¯1.5 .4 ¯.3
2 ¯1 0 0
```

Paralleling the floor function is the *ceiling* function; `⌈X` yields the smallest integer that is not smaller than X. Thus,

```
      ⌈2.3
3
      ⌈1.9
2
      ⌈2 ¯1.5 .4 ¯.3
2 ¯1 1 0
```

The formula `⌈¯.5+X` rounds X to the nearest integer, with numbers having fractional part of .5 being rounded to the next smaller integer:

```
      ⌈¯.5+2.3
2
      ⌈¯.5+1.9
2
      ⌈¯.5+2 ¯1.5 .4 ¯.3
2 ¯2 0 0
```

Floor and ceiling are monadic scalar functions; the argument of floor or ceiling may be any numeric array. For example,

```
      M                    ⌊M                   ⌊.5+M
1.5 ¯0.3              1 ¯1                 2 0
2    0.8              2  0                 2 1
```

5.10 Example: Rounding to a Decimal Position

Rounding in any decimal position can be accomplished by temporarily making the position of interest the units position. To round X to hundredths, apply a rounding formula to $100\times X$, then move the decimal point back again: $.01\times\lfloor .5+100\times X$. An entire array of numbers may be rounded simultaneously. For example, let $V\leftarrow$ `.034 ¯1.0127 .0049 57.006 ¯.997 2`, then

```
      ⌊.5+100×V                    Round 100×V to nearest integer.
3 ¯101 0 5701 ¯100 200

      .01×⌊.5+100×V                Move decimal point back.
0.03 ¯1.01 0   57.01 ¯1  2
```

Similarly, a way to round the entries of an array X to whole thousands is to use `1000×⌊.5+.001×X`:

```
      V←123456 7499 7500 345 ¯602
      1000×⌊.5+.001×V
123000 7000 8000 0 ¯1000
```

APL provides a formatting function that can be used for rounding and other matters related to the form of data output. See Section 10.15 and Appendix B.

5.11 Membership Function

The *membership* function tests whether each element of its left argument is an element of its right argument; the arguments may be any arrays. The function is denoted by epsilon, ϵ. The result of $A\,\epsilon\,B$ (read as "A belongs to B" or "A in B") is a Boolean array having the shape of A with a `1` in each position where the corresponding entry of A is one of the entries of B. For example,

```
      3 6ε6 1 8          6 1 8ε3 6          (2 4⍴⍳8)ε3 6
0 1                  1 0 0                0 0 1 0
                                          0 1 0 0

      3 3 7 4ε⍳6         3 3 7 4ε3 2⍴⍳6         1 2 3ε4
1 1 0 1              1 1 0 1              0 0 0
```

Notice that the shape of the right argument is not relevant to the result of a membership test.

If V is a vector of data, the membership function gives a quick, easy test of whether the vector contains specified entries. Moreover, the expression $\wedge/U\,\epsilon\,V$ tests whether every element of vector U is in V; this can be interpreted as a test of set containment for the sets of distinct elements in vectors U

and V. To determine whether the collections of distinct elements in U and V are the same can be accomplished simply with $\wedge/(V\in U),U\in V$. (Notice the use of catenation here.)

Here are some membership function examples:

(a) Find the sum of those numbers in vector U that are also in vector V. Solution: $+/U\times U\in V$ or, equivalently, $U+.\times U\in V$.

(b) Find the sum of the numbers in vector U that are not in vector V. Solutions: (i) $(+/U)-U+.\times U\in V$, (ii) $U+.\times\sim U\in V$.

5.12 Comparison Tolerance

If print precision, $\square PP$, is set to the maximum value for an APL system, the displayed value of $3\times1\div3$ is seen to be not 1 but very close to 1. Yet when APL tests equality in $1=3\times1\div3$, the result is (as seems reasonable) 1, indicating true, even though the machine evaluations of the two sides differ. APL evaluates $A=B$ as true if the computer evaluations of A and B are relatively close. The measure of relative closeness involves the system variable *comparison tolerance*, $\square CT$; it is assignable by the user but in a clear workspace is typically on the order of $1E^{-}14$. Precisely, APL evaluates $A=B$ as true if the computer evaluation of $|A-B$ is less than the evaluation of $\square CT$ times the larger of $|A$ and $|B$; thus, $A=B$ is evaluated as true if the magnitude of their difference divided by the larger of their magnitudes is less than $\square CT$.

Ceiling, floor, membership, and the relational functions are all sensitive to $\square CT$. For example,

```
      □CT←1E¯14
      1=3×1÷3                 ⌊3×1÷3                  1>3×1÷3
1                       1                       0

      □CT←0
      1=3×1÷3                 ⌊3×1÷3                  1>3×1÷3
0                       0                       1
```

The test of equality based on relative closeness can give results that might be unexpected on first meeting. For example,

```
      U←0 1E¯15 1

      □CT←1E¯14                       □CT←0
      0=U                             0=U
1 0 0                           1 0 0

      1=1+U                           1=1+U
1 1 0                           1 0 0
```

That $0=1E^{-}15$ yields 0 while $1=1+1E^{-}15$ yields 1 if $\square CT$ is $1E^{-}14$ may seem surprising, but these are valid consequences of the definition of the equals function.

Computers typically do not represent numbers exactly, and working near the limits of any machine's capacity to distinguish numbers that are very close can be problematical. Consider this example that was processed by one particular system:

```
      □CT←0
      □PP←16
      V←¯1E¯16 1E¯16 ¯2E¯16 2E¯16
      1+V
0.9999999999999999  1  0.9999999999999998  1
      1=1+V
0 1 0 1
```

Exercises

1. Write an expression that gives the number of nonzero entries in vector V.
2. Write expressions to count the entries of vector V that are (a) more than, (b) less than, D units from some fixed number M.
3. Let V be a vector; express verbally the result of each expression.
   (a) `2=V` **(b)** `2 5 9∘.=V` **(c)** `∨≠2 5 9∘.=V` **(d)** `+/∨≠2 5 9∘.=V`
   Check your answers with `V←9 9 2 5 4 ¯5 9 2 3`.
4. Write an expression to count the entries of vector V that are in at least one of the intervals (J,K) or (M,N).
5. Write an expression to count the entries of vector V in the intersection or overlap of intervals (J,K) and (M,N).
6. What information does the expression

   `+/∨≠((J,M,P)∘.<V)∧(K,N,Q)∘.>V`

   yield about the entries of vector V relative to intervals (J,K), (M,N), and (P,Q)?
7. Write an expression to count the number of integer entries in vector V.
8. Let V be a vector; express verbally the result of each expression.
   (a) `V∈2` **(b)** `V∈2 5 9` **(c)** `+/V∈2 5 9`
   Check your answer with `V←9 9 2 5 4 ¯5 9 2 3`. Compare your answer to that for Exercise 3.
9. Let A be any array and P be a positive number. Write an APL expression that replaces all entries of A having absolute value smaller than P by zeros but does not affect other entries of A.
10. With S a matrix, write an expression to compute the averages of the numbers in each row of S, not counting any number smaller than 10. Test your expression on

```
      S
20 22 19 23 21 16 18
 3 15 17 14 18 16 13
18 15 12 19 13  8 14
16 14 10  5 15 12 15
 4  7 14 10 16 12 14
20 21  0 19 22 18 20
```

11. Let V be a vector having as entries the ages in years of the people in a certain population. Write an expression to give the frequency distribution of this data in the intervals [0, 20], (20, 35], (35, 50], (50, 65], and (65, 120].

12. Express verbally the condition tested by these inner products if U and V are equal length vectors:

(a) $U\wedge .=V$ **(b)** $U\wedge .\neq V$
(c) $U\wedge .<V$ **(d)** $U\vee .\leq V$
(e) $U\vee .\neq V$ **(f)** $1\vee .=V$
(g) $0\wedge .<V$

13. Observe the error messages produced by

(a) `1 2+.×1 2 3` **(b)** `2 5≠1 4 4`
(c) `~2` **(d)** `1⍲2`
(e) `0⍱4` **(f)** `(2 3⍴0)+.×2 3⍴0`

Note: At least one APL system has generalized the functions $\wedge$ and $\vee$: $A\vee B$ yields the greatest common divisor of A and B; $A\wedge B$ yields the least common multiple of A and B.

14. A student's course grades over three terms are

Course grade	B	A	B	B	C	B	A	B	D	A	A	C	B	B	C	B
Credit hours	3	3	3	4	3	3	3	4	4	1	3	3	4	3	3	1

with A, B, C, D having values 4, 3, 2, 1, respectively. Assign the data to two vectors and use those vectors to compute the student's grade point average.

15. (Continuation of Example 5.8.) If instead of just testing whether there is at least one path of length 2 from vertex i to j. you wished to count these paths, you can do so by counting the k's, $k = 1, 2, \cdots, n$, for which the kth entries of row i and column j are both 1s. What inner product of A with itself will do that?

16. Suppose vector S contains a list of scores, each score being in the range from 0 to 650. Write an expression that computes the percentage that each score is of the maximum score of 650 expressed to the nearest tenth of a percentage point.

17. (For readers who know relations on sets.) If A, B, and C are finite sets, R is the (Boolean) matrix of a relation r from A to B, and S is the (Boolean) matrix of a relation s from B to C, show the matrix of the composition of r and s is given by the inner product $R\vee .\wedge S$.

In the special case with $A = B = C$ and $r = s$, the composition of r with itself has matrix $R\vee .\wedge R$, call it $R2$. Similarly, the k-fold composition of R with itself has matrix

$$Rk\leftarrow R\vee .\wedge R\vee .\wedge \cdots \vee .\wedge R \quad (k \text{ factors.})$$

Using this notation, the matrix of the connectivity relation determined by r (also called the transitive closure of r) is given by $R\vee R2\vee R3\vee \cdots \vee Rn$ where n is the number of elements of A.

18. (For readers who know the method of Warshall [W].) Write an APL function that generates the transitive closure of a matrix of a connectivity relation by the method of Warshall.

6

Simulation and More Mathematical Functions

This chapter introduces the random selection functions of APL, roll and deal. It illustrates their use with probability simulations and Monte Carlo integrations. The chapter also introduces the factorial and binomial functions, with applications to probability problems, including the binomial probability distribution, and a polynomial translation formula. Finally, it presents the circular, hyperbolic, and pythagorean trigonometric functions and multiplication by pi. The user-defined functions in this chapter use the relationals from Chapter 5 and are a little more complex than previous examples.

6.1 Roll, Deal, and Random Link

Random selections can be made with the APL function *roll,* which is named in analogy with a roll of a die. Roll is denoted by ? and takes positive integers as arguments; ?*N* selects one element from ⍳*N* with all elements being equally likely. For example,

```
      ?6
1

      ?⍳5
1 1 2 4 1
```

```
      ?6 6
5 3

      ??10
1
```

```
      ?5⍴6
4 2 1 5 5

      ?2 5⍴10
7 1 4 1 5
7 6 10 9 6
```

As an illustration of roll, consider the expression `(S<4)∨10<S←+/?6 6`, which yields 1 if the total points on a simulated roll of two dice is either greater than 10 or less than 4 and returns a 0 otherwise.

Note that roll is a scalar monadic function. The dyadic use of ? signifies the APL function *deal*, named in analogy with a deal of a hand from a deck of cards. For example, `3?10` would select a vector of three distinct elements from `⍳10`, with all such triplets being equally likely. For positive integers M and N with $M \leq N$, the expression $M?N$ yields a random selection without replacement of M elements from ιN. The special case $0?N$ yields the empty vector $\iota 0$. Examples are

```
      3?8          3?8          8?8
8 3 4        3 1 2        6 7 2 4 8 1 3 5

      1?10         1?10         2?1
8            1                        (Error message.)
```

Deal is not a scalar function; it does not accept higher dimensional arrays in either argument.

The value of $?N$ is determined by two numbers: N and the APL system variable *random link*, denoted $\square RL$. Each execution of roll causes $\square RL$ to cycle to another value in a large set of positive integers. Before any execution of ? in a clear workspace the value of $\square RL$ is `16807`. Random link is assignable by the user. Notice the changes in value of $\square RL$ after uses of roll or deal:

```
      ⎕RL←16807    ⎕RL←16807    ⎕RL←16807
      ?6           ?5⍴9         5?9
1            2 7 5 5 2    8 2 4 3 7
      ⎕RL          ⎕RL          ⎕RL
282475249    470211272    470211272
```

It is advisable to record the value of $\square RL$ at the outset of any sequence of computations involving roll or deal that you might wish to review or duplicate. When you specifically do not want to repeat a prior set of "random" choices, assign $\square RL$ a different value at the outset of each such work block. Random link can be made a local variable in a defined function; in that case, it must be assigned a value before any execution of roll or deal.

6.2 *Example: Matching Partners*

Hank and Sally Stepintime, cochairs of the local chapter of SYPSDA (Swing Your Partner Square Dancers of America), enlivened the weekly dances for their 20 dancing couples with a random partner drawing for one of the dances. Slips of paper with the men's names were drawn out of a hat by the women to determine their partners for the dance. More often than not, at least one woman would draw her own escort's name. Many executions of a simulation of this drawing scheme suggest that such coincidences, or matches, are to be expected.

The random partner drawing can be simulated with deal. Let the men be represented by the numbers 1–20 in the vector `ESCORTS←⍳20`. The drawing of partners will be represented by `PARTNERS←20?20`. Then `PARTNERS=ESCORTS` will show any coincidence of partners

and escorts, and `+/PARTNERS=ESCORTS` will give the count of such coincidences:

```
      □RL←1000
      ESCORTS←⍳20
      □←PARTNERS←20?20
20 9 8 6 4 1 14 2 19 16 5 17 15 3 11 13 12 18 10 7
      PARTNERS=ESCORTS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
      +/PARTNERS=ESCORTS
1
```

Running this simulation and counting coincidences several times, but omitting the variable names gives the following:

```
      +/(20?20)=⍳20          +/(20?20)=⍳20          +/(20?20)=⍳20
0                      1                      1

      +/(20?20)=⍳20          +/(20?20)=⍳20          +/(20?20)=⍳20
3                      5                      1
```

Students of probability might recall that the probability of at least one coincidence (match) between partner and escort is approximately $1 - 1/e \approx 0.63$.

6.3 *Example: Simulating a Classical Probability Experiment*

Each of three boxes contains two balls, one ball in each of two compartments in each box. One of the boxes contains two red balls, one has one red and one white ball, and one has two white balls. Suppose a box and one of its compartments is chosen at random and the ball therein is seen to be red. What is the probability that the ball in the other compartment is red?

The answer, which is not 1/2, can be calculated with aid of conditional probabilities. Constructing APL simulations of the experiment is instructive, however, and numerical output can strengthen your intuition about conditional probabilities. The first simulation presented here adheres rather closely to the description of the experiment.

Let the rows of the 3-by-2 matrix

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}$$

correspond to the boxes, and let the columns determine compartments in the boxes. Signify the red and white balls by 1s and 0s, respectively. The function `S I M` below simulates performing the experiment `N` times; it returns the number of times when the ball in the other compartment is red, given that the first selected ball is red. Since, often the first selected ball is not red, drawings are simulated until red occurs on the first draw `N` times.

In line 1 the counters for `R`, the number of times the first draw is red, and `Z`, the number of times red is found in the other compartment after a red first draw, are initialized and the matrix `M` is defined.

A box *B* and compartment *C* are randomly chosen in line 2, thereby selecting the first ball. The selected ball is tested for being red in line 3 by the equality `1=M[B;C]`. If this is true, the branch is forward to line 4; if false, the branch is back to line 2 and another random selection of a first ball. Note that neither *R* nor *Z* is incremented if the first selected ball is not red. In line 4, *R* is incremented, and `M[B;1+C=1]` tests whether the ball in the "other" compartment is red, since `1+C=1` is the number of the "other" compartment. If that ball is red, *Z* is incremented. In line 5 the relation `R<N` tests whether *N* simulations have been completed. If not, the branch is back to line 2; if so, execution of *SIM* is terminated by a branch to zero. The function *SIM* is as follows:

```
     ∇ Z←SIM N;B;C;M;R
[1]    Z←R←0 ◊ M←3 2⍴1 1 1 0 0 0
[2]    B←?3 ◊ C←?2
[3]    →2+2×1=M[B;C]
[4]    R←R+1 ◊ Z←Z+M[B;1+C=1]
[5]    →2×R<N
     ∇
```

A few runs of *SIM* suggest that the desired probability is between .65 and .7 and probably close to .67:

```
      ⎕RL←16807
      SIM 10                  SIM 100                 SIM 1000
7                        65                      669
```

The function *SIM* was designed to mirror the experiment closely. It works; it does not, however, follow some recommended programming practices. It does not make the best use of APL and is slow.

Contrast *SIM* with the following simulation in which the loop with counter *R* in *SIM* is effectively replaced by use of the reshape function and the first selection is restricted to red balls. By randomly choosing boxes and compartments, each of the six balls is equally likely to be the first chosen one. This suggests ordering the balls in a vector `V←1 1 1 0 0 0` in which successive pairs of components [1, 1], [1, 0], [0, 0] represent the three boxes. Since the experiment is concerned only with cases for which the first chosen ball is red, a random choice need be made from among only the first three components of *V*. After that choice, the ball in the other compartment will be red if and only if the first chosen ball was in the first or second component of *V*. Thus, with `?3` indicating the position of the first selected ball, the other ball is red if and only if `2≥?3`. By this analysis *N* experiments can be simulated by executing `+/2≥?N⍴3`, which runs very rapidly:

```
      ⎕RL←16807
      +/2≥?100⍴3                      +/2≥?1000⍴3
62                              676
```

Since each execution of `?3` gives a sample from a uniform distribution on the set {1, 2, 3} and the other ball is red if and only if `2≥?3`, you can conclude that the desired probability is exactly 2/3.

6.4 *Example: Polya's Urn Scheme*

Some ideas of the previous example are extended in simulating a famous scheme of G. Polya. An urn contains *r* red balls and *w* white balls. A ball is drawn at random, its color is noted, and it and *k* more balls of the same color are added to the urn. The process is repeated. After each drawing, the ball

drawn and k balls of that color are added to the urn. What is the probability that the second ball drawn is red? The third ball? The fourth?

According to W. Feller [F, p. 83], "This scheme was devised for the analysis of phenomena like contagious diseases, where the occurrence increases their future probabilities".

The function *POLYA* below simulates four successive drawings made in accordance with Polya's scheme. Moreover, it allows *N* independent simulations of each drawing to be accomplished by applying the scheme to *N* urns simultaneously. The vector left argument of *POLYA* has *r*, *w*, and *k* as components; the right argument *N* specifies the number of repetitions (number of urns) of the scheme to be used. The fractions of the *N* times that red balls result on each of the first, second, third, and fourth drawings are reported in the output vector of the function. These fractions serve as estimates for the required probabilities.

In line 2 of *POLYA*, the initial total number of balls in each urn is stored in *T*, the loop counter for the number of drawings is initialized, and the output vector is set to the empty vector. Line 3 tests whether four drawings are completed. At each drawing from a given urn with *R* red balls and *T* (total) balls, when *R*≥?*T* yields 1 it represents drawing a red ball from the urn. In line 4, the relation *R*≥?*N*ρ*T* simulates *N* drawings simultaneously, and the results of the *N* drawings are stored in the *N*-element Boolean vector *B*. The components of *B* are averaged, and this fraction is adjoined to the output vector *Z*. Before the next drawing, the (differing) numbers of red balls in each urn and the total (common) number of balls in every urn is adjusted in line 5 using the outcomes of the previous drawing, which were recorded in *B*. Note that *R* becomes an *N*-element vector during the first execution of line 5. Line 6 marks the end of the loop. Here is *POLYA*:

```
     ∇ Z←V POLYA N;B; I;K;R;T;W
[1]    R←V[1] ◊ W←V[2] ◊ K←V[3]
[2]    T←R+W ◊ I←0 ◊ Z←⍳0
[3]    →4×I<4
[4]    B←R≥?Nρ T ◊ Z←Z,(+/B)÷N
[5]    I←I+1 ◊ R←R+K×B ◊ T←T+K
[6]    →3
     ∇
```

Simulations will be run with $r = 3$, $w = 7$, and k values 2 and 5; the fraction of reds on the first drawing should be near $r/(r + w)$, or .3. Examples follow:

```
      ⎕RL←16807
      3 7 2 POLYA 10
0.3 0.4 0.1 0.1

      3 7 2 POLYA 100
0.3 0.24 0.29 0.3

      3 7 2 POLYA 1000
0.298 0.303 0.306 0.295
```

```
      3 7 5 POLYA 10
0.2 0.3 0.2 0.3

      3 7 5 POLYA 100
0.32 0.34 0.23 0.3

      3 7 5 POLYA 1000
0.302 0.295 0.302 0.308
```

With many repetitions of this simulated experiment the fraction of red balls on each drawing is close to .3. These results suggest that the probability of drawing a red ball at any particular stage is independent of the number of drawings! This is, in fact, true; moreover, the probability, which is $r/(r + w)$, is independent of k, the number of balls added at each step.

6.5 *Factorial and Binomial*

For nonnegative integral arguments, the APL *factorial* function is the same as the factorial function of mathematics; thus, `! N` has the same result as `×/⍳N`. In particular, `!0` yields `1`. For example,

```
      !7
5040
      !⍳4
1 2 6 24
      !2 3⍴⍳6
 1  2  6
24 120 720
```

The infinite series $1 + 1/1! + 1/2! + 1/3! + \cdots$ has sum e, the base of the natural logarithm system. An APL expression for the sum of the first ten terms of this series, `+/÷!¯1+⍳10`, yields `2.718281526`, which is correct to the millionths place.

Recall the mathematical notation for the binomial coefficients:

$$\binom{n}{m} = \frac{n!}{m!\,(n-m)!}$$

Here the coefficients are used in a familiar formula:

$$(a+b)^3 = \binom{3}{0}a^3b^0 + \binom{3}{1}a^2b^1 + \binom{3}{2}a^1b^2 + \binom{3}{3}a^0b^3$$

The *binomial* (or combinatorial) function of APL is indicated by dyadic use of the factorial symbol. For example, the binomial coefficient $\binom{5}{2}$ is written as `2!5`:

```
      2!5
10
      3!5 4 8
10 4 56
      (0,⍳5)!5
1 5 10 10 5 1
```

```
      V←0,⍳5
      V∘.!V
1 1 1 1 1 1
0 1 2 3 4 5
0 0 1 3 6 10
0 0 0 1 4 10
0 0 0 0 1 5
0 0 0 0 0 1
```

```
      ⌽V∘.!V
1 0 0 0 0 0
1 1 0 0 0 0
1 2 1 0 0 0
1 3 3 1 0 0
1 4 6 4 1 0
1 5 10 10 5 1
```

Notice Pascal's triangle.

The binomial `M!N` has a combinatorial interpretation: If `M` and `N` are nonnegative integers, `M!N` yields the number of different subsets of `M` elements in a set of `N` elements; alternatively, it is the number of ways to choose `M` elements from `N` elements. For example, the probability of getting 2 aces in a bridge hand is the product of the number of ways to choose 2 aces from the 4 aces in the deck, times the number of ways to choose the other 11 cards in the hand from the 48 non-aces, divided by the number of ways to choose 13 cards from the deck of 52, or `(2!4)×(11!48)÷(13!52)`, which yields `0.2134933974`. There is more about the factorial and binomial functions in the exercises.

6.6 *Example: Binomial Distribution*

The probability function called the binomial distribution (or binomial density function) measures the probability of achieving k successes in n independent trials of an experiment with constant success

probability p. A random variable X with a binomial distribution has probability function given by

$$P(X = k) = \begin{cases} \binom{n}{k} p^k (1 - p)^{n-k} & \text{for } k = 0, 1, \cdots, n \\ 0, \text{ all other } k \end{cases}$$

Here is a typical type of use of a binomial distribution: If a production run of a large number of widgets is assumed to have 3% defectives, what is the probability that a random collection of 20 widgets contains 2 defectives? Viewing the sampling of 20 from a "large number" as being independent trials and applying the binomial distribution with success meaning the choosing of a defective widget, the required probability is `(2!20)×(.03*2)×.97*18`, or `0.09882966589`.

Binomial distribution problems often inquire about the probability of at most k successes in n trials or at least k successes in n trials. The defined functions ***BINATMOST*** and ***BINATLEAST***, respectively, compute such probabilities. The left argument of each function is a vector containing the parameters n and p; the right argument is the maximum (minimum) number of successes, and the output is the probability of at most (least) k successes in n independent trials with constant success probability p. In ***BINATMOST*** the probabilities of exactly $0, 1, \cdots, k$ successes are summed to get the desired probability; correspondingly, the probabilities of $k, k+1, \cdots, n$ successes are summed in ***BINATLEAST***:

```
     ∇ Z←NP BINATMOST K;A;N;P
[1]    N←NP[1] ◊ P←NP[2] ◊ A←0,⍳K
[2]    Z←+/(A!N)×(P*A)×(1-P)*N-A
     ∇

     ∇ Z←NP BINATLEAST K;A;N;P
[1]    N←NP[1] ◊ P←NP[2] ◊ A←K,K+⍳N-K
[2]    Z←+/(A!N)×(P*A)×(1-P)*N-A
     ∇
```

Returning to the widgets example above, the probabilities of getting at most 2 and at most 1 defectives in a batch of 20 are given by

```
      20 .03 BINATMOST 2
0.9789916436
      20 .03 BINATMOST 1
0.8801619777
```

And the probabilities of at least 2 and at least 1 defectives in a batch of 20 are

```
      20 .03 BINATLEAST 2
0.1198380223
      20 .03 BINATLEAST 1
0.4562056571
```

This last example can be alternatively computed as 1 minus the probability of no defectives, or $1 - P(X = 0)$, which is evaluated by `1-(0!20)×(.03*0)×.97*20` or simply `1-.97*20`, which yields `0.4562056571`. The relation between at most and at least is further exploited in Exercise 10.

6.7 Example: Polynomial Translation

Given a polynomial such as $P(t) = 3 - 4t + 2t^2$, the question arises: What are the coefficients of the powers of t for a composite function such as $P(t + 5)$? For this example, it is easy to check that $P(t + 5) = 3 - 4(t + 5) + 2(t + 5)^2 = 33 + 16t + 2t^2$, and the required coefficients are 33, 16,

and 2. This is an example of finding the coefficients of a translated polynomial $P(t + d)$ for a given P and translation d.

In general let

$$P(t) = \sum_{k=0}^{n} p_k t^k$$

Then using the binomial formula

$$P(t + d) = \sum_{k=0}^{n} p_k (t + d)^k = \sum_{k=0}^{n} p_k \sum_{i=0}^{k} \binom{k}{i} t^i d^{k-i}$$

and by changing the order of summation

$$P(t + d) = \sum_{i=0}^{n} t^i \sum_{k=i}^{n} p_k \binom{k}{i} d^{k-i}$$

This last summation gives formulas for the desired coefficients.

An efficient APL function for generating these coefficients is developed next. For notational simplicity consider this last formula with $n = 3$; then

$$\begin{aligned} P(t + d) = t^0 \Bigg[& p_0 \binom{0}{0} d^0 + p_1 \binom{1}{0} d^1 + p_2 \binom{2}{0} d^2 + p_3 \binom{3}{0} d^3 \Bigg] \\ + t^1 \Bigg[& \cdot \qquad p_1 \binom{1}{1} d^0 + p_2 \binom{2}{1} d^1 + p_3 \binom{3}{1} d^2 \Bigg] \\ + t^2 \Bigg[& \cdot \qquad \cdot \qquad p_2 \binom{2}{2} d^0 + p_3 \binom{3}{2} d^1 \Bigg] \\ + t^3 \Bigg[& \cdot \qquad \cdot \qquad \cdot \qquad p_3 \binom{3}{3} d^0 \Bigg] \end{aligned}$$

The vector of coefficients of powers of t in this equation can be seen as the `+ . ×` inner product of the matrix

$$\begin{bmatrix} \binom{0}{0} d^0 & \binom{1}{0} d^1 & \binom{2}{0} d^2 & \binom{3}{0} d^3 \\ 0 & \binom{1}{1} d^0 & \binom{2}{1} d^1 & \binom{3}{1} d^2 \\ 0 & 0 & \binom{2}{2} d^0 & \binom{3}{2} d^1 \\ 0 & 0 & 0 & \binom{3}{3} d^0 \end{bmatrix}$$

and the vector with components p_0, p_1, p_2, p_3. This latter matrix is a simple `×` product of two matrices generated from `V←0 1 2 3` — namely, `V∘.!V` and `D*⍉V∘.-V`. For clarity, only the expo-

nent portion of the last matrix will be displayed; putting in a value of *D* would obscure the structure. Notice that the positions of the negative entries in ⍉*V* ∘ . −*V* correspond to zero entries of the matrix *V* ∘ . ! *V* and in the multiplication the negative powers of *D* will vanish:

```
      V←0 1 2 3
      V ∘ . ! V
1 1 1 1
0 1 2 3
0 0 1 3
0 0 0 1

                ⍉V ∘ . -V
 0  1  2  3
-1  0  1  2
-2 -1  0  1
-3 -2 -1  0
```

The left argument of the function ***POLYTRANSLATE*** defined below is the coefficient vector of a given polynomial *P* in order of increasing powers with 0s for missing powers, and the right argument is the scalar shift *d*. The result is the vector of coefficients of the polynomial translation $P(t + d)$:

```
     ∇ Z←P POLYTRANSLATE D;V
[1]    V←¯1+⍳⍴P
[2]    Z←((V ∘ . ! V)×D*⍉V ∘ . -V)+ . ×P
     ∇

      8 0 ¯3 1 2 POLYTRANSLATE ¯1
6 1 6 ¯7 2
```

The last example shows that $6 + t + 6t^2 - 7t^3 + 2t^4$ is the polynomial translation of $8 - 3t^2 + t^3 + 2t^4$ with t replaced by $t - 1$.

6.8 *Trigonometric Functions*

The circular and hyperbolic functions are primitive APL functions, and, as with all the primitives, the trig functions are invoked with special symbols. (Letter combinations such as ***SIN*** or ***COS*** remain available for user-defined functions or variables.) The APL *circle*, ○, is used dyadically for this family of functions; for example, the cosine of *X* is given by 2○*X*. Angle measures are in radians. The circle functions defined with square roots are known as pythagorean functions. Here is a list of corresponding notations:

$$\sqrt{1 - X^2} \leftrightarrow 0\circ X$$

$\sin X \leftrightarrow 1\circ X$	$^-1\circ X \leftrightarrow \arcsin X$
$\cos X \leftrightarrow 2\circ X$	$^-2\circ X \leftrightarrow \arccos X$
$\tan X \leftrightarrow 3\circ X$	$^-3\circ X \leftrightarrow \arctan X$
$\sqrt{1 + X^2} \leftrightarrow 4\circ X$	$^-4\circ X \leftrightarrow \sqrt{^-1 + X^2}$
$\sinh X \leftrightarrow 5\circ X$	$^-5\circ X \leftrightarrow \text{arcsinh } X$
$\cosh X \leftrightarrow 6\circ X$	$^-6\circ X \leftrightarrow \text{arccosh } X$
$\tanh X \leftrightarrow 7\circ X$	$^-7\circ X \leftrightarrow \text{arctanh } X$

Circle used monadically, as in ○X, yields pi times X. Thus, ○1 is 3.141592654, and ○30÷180 is 0.5235987756, which is the radian equivalent of 30 degrees angle measure; for example,

```
      1○0                    2○0                    ¯2○1
0                      1                      0
      1○0 30÷180             1○0÷6                  (180÷○1)×¯1○0.5
0.5                    0.5                    30
```

6.9 *Example: Monte Carlo Integration*

Let f be a continuous function satisfying $0 \leq f(x) \leq M$ on $[a, b]$. One approach to estimating $\int_a^b f(x)\,dx$ is to multiply $M(b - a)$ by the fraction of a set of points randomly distributed in the rectangle $\{(x, y) \mid a \leq x \leq b,\ 0 \leq y \leq M\}$ that lies below the graph of f. The idea extends to more general regions and integrals. In applying the technique, the defined function *RANREAL* introduced in Section 3.4 will provide random choices of *N* points from the interval [0, 1]:

```
     ∇ Z←RANREAL N
[1]    Z←1E¯17×?Nρ1E17
     ∇

      □PP←5
      RANREAL 1              RANREAL 2              RANREAL 2 2
0.13154                0.45865 0.21896        0.67886 0.93469
                                              0.51942 0.034572
```

For a first example, consider the familiar integral

$$\int_0^{\pi/2} \cos x\,dx$$

(Caution: ○.5 corresponds to $\pi/2$, although it looks like 0.5.)

```
      X←(○.5)×RANREAL 1000      Random x's on [0, π/2].
      Y←RANREAL 1000            Random y's on [0, 1].
      Q←.001×+/Y<2○X            Fraction of points below graph.
      ○.5×1×Q                   Estimate of integral.
1.0053
```

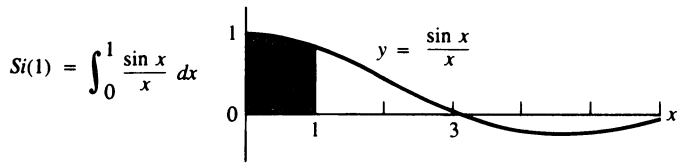


Figure 6.1. The sine-integral of 1.

Next, estimate the sine integral (Si) of 1, shown in Figure 6.1:

```
      X←RANREAL 1000
      Y←RANREAL 1000          Note (sin x)/x lies in [0, 1].
      .001×+/Y<(1○X)÷X        Estimate of integral Si(1).
0.931
```

The area inside the unit circle $x^2 + y^2 = 1$ in the first quadrant is $\pi/4$. Hence, an approximation of that area multiplied by 4 approximates π. Note the use of the pythagorean function 0○ in this example:

```
      ⎕RL←16807
      X←RANREAL 10000 ◊ Y←RANREAL 10000
      (4÷10000)×+/Y≤0○X
3.1592
```

Finally, estimate $\iint_R \cos(xy)\, dA$, where R is the triangular region in the first quadrant of the x–y plane having vertices (0, 0), (1, 0), and (0, 1). For (x, y) in R, $0 < \cos(xy) \leq 1$; thus, the relevant part of the graph is in the triangular prism having base R and height 1. The prism in turn is a subset of the cube $\{(x, y, z) \mid 0 \leq x, y, z \leq 1\}$. A sample from a uniform distribution of points over this cube will be generated, and the fraction of points (x, y, z) with (x, y) in R *and* z less than $\cos(xy)$ will be multiplied by the volume of the cube, 1, to estimate the integral. Note that (x, y) is in R corresponds to $y < 1 - x$:

```
      X←RANREAL 5000
      Y←RANREAL 5000
      Z←RANREAL 5000                The points (x, y, z) are in the cube.
      .0002×+/(Z<2○X×Y)∧Y<1-X       Approximates the integral.
0.4944
```

Exercises

1. Guess the most likely value of `????10`. Execute the expression several times. Try `????100⍴10`.
2. Write an expression that simulates a roll of three dice and returns a `1` if any die shows a 6 or if the sum of the points on the three dice is less than 5, and returns a `0` otherwise.
3. In Example 6.2, the counting of matches in a drawing was simulated by the expression `+/(N?N)=⍳N` with `N←20`. Experiment with this simulation with `N` values `5`, `50`, `100`, `1000`. Do matches seem likely in all cases?
4. Define a function, including a loop, that simulates `N` drawings to approximate the average number of matches of escorts and partners in Example 6.2. More formally, if X is the random variable giving the number of matches as a result of a drawing of the names from the hat, estimate $E(X)$, the expectation of X. Try the function with `N←100`.
5. (Refer to Example 6.3.) Suppose there are four boxes: one having two red balls, two having one red and one white ball, and one having two white balls. Given that the first randomly selected ball is red, what is the probability that the other ball in the same box is red? This experiment can be simulated with a defined function like `SIM`, but that is not recommended here. Instead, since each ball is equally likely to be drawn first, think of numbering the red balls from 1 to 4, with

balls numbered 1 and 2 in the same box. Let ?4 indicate the first selected red ball, then ··· (Finish the simulation of counting the number of other-ball-is-red outcomes in *N* drawings.) From your analysis determine the exact probability.

6. Modify *POLYA* of Section 6.4 so that an arbitrary number, *D*, of drawings may be simulated. Begin by making the right argument of *POLYA* a two-element vector containing *D* and *N*.

7. Describe in words the variation on Polya's urn scheme (Section 6.4) that is simulated by the function *NEWPOLYA*:

```
     ∇ Z←V NEWPOLYA N;B;I;R;T;W
[1]    R←V[1] ◊ W←V[2]
[2]    T←R+W ◊ I←0 ◊ Z←⍳0
[3]    →4×I<4
[4]    B←R≥?N⍴T ◊ Z←Z,(+/B)÷N
[5]    R←R+B--B ◊ W←W+(~B)-B
[6]    ⎕←I←I+1
[7]    →3×∧/0≠R,W
     ∇
```

Note that this function can stop execution and return a result before four draws are completed if the number of red or white balls is initially less than 4.

8. The APL factorial function !*X* corresponds to the gamma function of mathematics evaluated at *X*+1. Try:
   (a) !¯1.5,¯.5,.5,(*1),○1 **(b)** !¯1 **(c)** !¯2

9. Experiment with the binomial function. Try 1.2!¯.75 and (*1)!○1. Try *V*∘.!*V*←¯4+⍳7 and *U*∘.!*U*←¯3.5+⍳6. But, try .5!¯1 and ¯.3!¯2 (first argument nonintegral, second argument a negative integer).

10. Redefine the function *BINATLEAST* of Section 6.6 in terms of the function *BINATMOST*.

11. Let the vector *X* be given by *X*←*RANREAL* 10 (see Section 6.8). Predict the output of the expression +/((1○*X*),2○*X*)*2.

12. Approximate $\int_0^1 \sqrt{1 + x^4}\,dx$ using the Monte Carlo technique with 1000 points.

13. Approximate $\int\int_R \sqrt{1 - x^2y^2}\,dA$, where *R* is the square $0 \le x, y \le 1$. Use 10000 points. (Suggestion: Use a pythagorean function.)

14. Most people are surprised to learn that in any randomly selected group of more than 22 people there is a greater than 50–50 chance that two people share the same birthday. For a group of 23, the probability is near .51 that there is a shared birthday. For 41 people the probability is greater than 90% that there is a shared birthday. The expression *B*←?23⍴365 generates a vector of 23 randomly chosen "birthdays". Incorporate that expression in a defined function with a loop that simulates sampling sets of 23 birthdays *N* times and reports the number of times there are any shared birthdays.

7

Statistics and Graphics

This chapter uses APL to analyze data. It introduces several primitive APL functions and operators to compute descriptive statistics. These provide the capability to find minimum and maximum values and the capacity to order, replicate, and accumulate data. The chapter also introduces character arrays, which are used to display data visually via histograms and scatter plots. Linear regression is recalled and data are plotted along with the line of best fit. The chapter ends with sampling from several types of distributions.

7.1 Mean and Standard Deviation

Given a list of data as a vector, the mean or average of the entries in the vector can be computed as in Section 3.1:

```
     ∇ Z←AVG W
[1]    Z←(+/W)÷⍴W
     ∇

       X←2 ¯1 4 7 7 1 8          Y←5 7 8 9 9 4 8 7 6 5 8 8

       AVG X                     AVG Y
4                              7
```

The sample standard deviation can be computed using

```
     ∇ Z←SD W
[1]    Z←((+/(W-AVG W)*2)÷¯1+⍴W)*0.5
     ∇

       X-AVG X
¯2 ¯5 0 3 3 ¯3 4

       +/(X-AVG X)*2
72

       SD X                              SD Y
3.464101615                       1.651445648
```

7.2 Maximum and Minimum

APL provides *minimum* and *maximum* functions, which are invoked by the dyadic use of the symbols ⌊ and ⌈; these symbols were used monadically for ceiling and floor in Section 5.9. Thus, 3⌊4 is 3, the minimum of 3 and 4. Likewise, 2⌊¯5 is ¯5 and 2⌊¯3⌊125⌊0 is ¯3. When *W* is a vector, the expressions ⌊/*W* and ⌈/*W* produce the minimum and maximum entries of *W*, respectively. Recall that ⌊/*W* gives the same result as inserting a ⌊ between all elements of *W*. Examples follow:

```
       A                    B                    C
1 3 1 4 7           ¯2 5 ¯5 4 6          4 2 3
                                         1 6 4

       ⌊/A                  ⌊/B                  ⌊/C
1                   ¯5                   2 1

       ⌈/A                  ⌈/B                  ⌈/C
7                   6                    4 6

       A⌊B                  A⌈B                  3⌈C
¯2 3 ¯5 4 6         1 5 1 4 7            4 3 3
                                         3 6 4

       ⌈/A⌊B                A⌈.⌊B
6                   6
```

7.3 Example: Range

The range of a data set given as a vector can be computed using the maximum function ⌈ and the minimum function ⌊:

```
     ∇ Z←RANGE W
[1]    Z←(⌊/W),⌈/W
     ∇
```

```
      X
2 ¯1 4 7 7 1 8

      RANGE X
¯1 8
```

```
      Y
5 7 8 9 9 4 8 7 6 5 8 8

      RANGE Y
4 9
```

7.4 Grade Up

The APL *grade up* function, ⍋, results in a list of indices. If a vector argument V has ***distinct*** entries, then ⍋V is the vector containing the index of the smallest element of the vector V, followed by the index of the next smallest element of V, and so on, up to the index of the largest element of V. When the vector V has ***repeated*** entries, the indices of the repeated entries appear in increasing order. For example,

```
      X
2 ¯1 4 7 7 1 8

      ⍋X
2 6 1 3 4 5 7

      X[⍋X]
¯1 1 2 4 7 7 8
```

```
      Y
5 7 8 9 9 4 8 7 6 5 8 8

      ⍋Y
6 1 10 9 2 8 3 7 11 12 4 5

      Y[⍋Y]
4 5 5 6 7 7 8 8 8 8 9 9
```

Notice the APL idiom V[⍋V] orders the elements of the vector V. This indirect method for ordering vectors allows considerable flexibility in ordering matrices. Further uses of ⍋ and ⍒ are considered in Sections 9.1 and 9.2.

7.5 Example: Median

The median of a data set is a point halfway up the ordered data. Once the data have been ordered, it is fairly easy to identify the median.

Consider a vector V of data. If ⍴V is odd, the median is the middle entry of V[⍋V]. If ⍴V is even, the median of V is taken to be the average of the two middle entries of V. Thus, the median of 1 3 4 7 8 is 4, and the median of 1 3 4 7 is 3.5.

The even and odd cases are treated together in the function *MEDIAN*. If ⍴W is odd, both ⌈.5×⍴W and ⌊1+.5×⍴W give the middle index of W. If ⍴W is even, ⌈.5×⍴W and ⌊1+.5×⍴W give the two middlemost indices of W. The average of the elements of the ordered vector having these indices is the median in either case. Let X and Y be as in the previous section:

```
     ∇ Z←MEDIAN W                    W is a vector of data.
[1]    W←W[⍋W]                       First order the data.
[2]    Z←.5×W[⌈.5×⍴W]+W[⌊1+.5×⍴W]
     ∇

      MEDIAN X
4
```

```
      MEDIAN Y
7.5
```

7.6 Character Arrays

Vectors of characters can be entered by enclosing characters within single quotation marks. Many APL functions that apply to numeric arrays also apply to character arrays. For example,

```
      X←'HI  JOE'
      X,Y
HI  JOE KELLEY
      ⍴X
6
      2 3⍴X
HI
JOE

      '*',(2 3⍴X),'*'
*HI  *
*JOE *
```

```
      Y←'  KELLEY'
      Y='E'
0 0 1 0 0 1 0
      +/Y='E'
2
      +/Y≠' '
6
```

```
      Z←'□*'
      Z[2 2 1 1 1 2]
**□□□*
      Z[1+Y='E']
□□*□□*□
      Z[1+1 2∘.<1 2 3]
□**
□□*
```

Notice that `+/Y='E'` counts the number of `E`'s in `Y` and `+/Y≠' '` counts the number of non-blank characters.

The quote character itself requires special consideration. Two single quotes consecutively indicate a quote mark within a character vector:

```
      'IT''S EASY'
IT'S EASY
```

7.7 Example: Frequencies

Consider a set `A`, called a *sample space*, from which some data `W` are taken with repetitions allowed. The number of times each element of `A` appears in `W` is called the *frequency* of that element in `W`:

```
      A←⍳8
      W←3 5 4 3 3 3 7 8 5 2 2 1
      A∘.=W
0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 1 1 0
1 0 0 1 1 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0
      FREQ←+/A∘.=W
      FREQ
1 2 4 1 2 0 1 1
```

Thus, there is one 1, two 2s, four 3s, and so on, occurring in the data *W*. Whenever *A* and *W* are vectors, the idiom +/*A* ∘ . =*W* gives a vector of the frequencies of the elements of *A* in *W*.

Section 9.6 considers another approach to computing frequencies. That approach is faster for large data sets but is more complicated than the method in this section.

7.8 *Example: Histograms*

Once a vector of the frequencies has been computed, it is fairly easy to display the frequencies with a histogram. First, a Boolean matrix is constructed. The number of 1s in the lower positions of a column is the corresponding entry in the vector of frequencies. Using the frequencies of the elements of *A* in *W* from above gives

```
      FREQ
1 2 4 1 2 0 1 1

      4 3 2 1∘.≤FREQ
0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0
0 1 1 0 1 0 0 0
1 1 1 1 1 0 1 1
```

This matrix gives a histogram of the data *W*.

Next, the matrix is used to construct a character matrix that is visually more appealing. One is added to those Boolean values to get indices for a character vector. The character vector has two entries, a blank and a star. The zeros in the Boolean outer product ∘ . ≤ correspond to blanks and the 1s correspond to the stars:

```
      C←' *'
      C[1+4 3 2 1∘.≤FREQ]
  *                       *
  *                       *
 **  *                    **  *
***** **                  ***** **
```

```
      ' *'[1+4 3 2 1∘.≤FREQ]
  *
  *
 **  *
***** **
```

Notice that there is no need to name the vector ' *' of characters used. The outer product and indexing the character vector ' *' will be used in a function *HIST*. The function takes integer data given in a vector as its argument *W* and results in a histogram. The sample space, *A*, is taken to be the integers within the range of the data. Notice that with *F* as the vector of frequencies, *M*+1−⍳*M*←⌈/*F* is a vector of consecutive integers decreasing from the maximum frequency *M*, down to 1:

```
     ∇ Z←HIST W;M;A;F
[1]    A←M+¯1+⍳1+(⌈/W)−M←⌊/W          Find the sample space.
[2]    F←+/A∘.=W                      Find the frequencies.
[3]    Z←' *'[1+(M+1−⍳M←⌈/F)∘.≤F]
     ∇
```

```
      HIST 3 5 4 3 3 3 7 8 5 2 2 1
  *
  *
 ** *
******  **
```

If data are not given as integers, you can round the data to be integral. Moreover, if the data are too spread out, you can scale them by a suitable factor. The situation of having too much data for a nice histogram is considered in the exercises. As an example of handling spread-out real data, consider the vector *Y*, which has 25 elements:

```
      5 5⍴Y
 5.00572017 15.91457254 20.81441836 20.02868071 24.61890713
27.15621801 28.83345066 30.18212575 29.95967286 29.26922668
35.37330529 33.67852318 36.66205049 35.97868324 36.78051286
42.47346322 38.44806    37.54378816 43.56616929 46.82608646
43.35768679 48.28296999 51.09287784 56.02385012 58.61443855

      HIST ⌊.5+Y                    Round the data, then use HIST.
                          **    **
*           *   **   * * **   *****   ***  **  *    *  *

      HIST ⌊.5+.2×Y                 Scale back by a factor of .2 first.
       *
      **
      ****
    *******
* **********
```

7.9 *Replicate on Vectors*

The *replicate* function, designated by a slash, /, can be used to select, discard, or repeat data. Since the left argument of replicate is a numeric vector, there is no ambiguity with the use of slash for the reduction operator, which has a function left argument. Here replicate is applied to vector right arguments; see Chapter Eight for extending replicate to arrays.

The arguments of the replicate function are vectors of equal length. The elements of the left argument are nonnegative integers indicating the number of times the corresponding item of data on the right is to be repeated:

```
      1 2 4 1/¯3 6 1 2              1 4 1 4 1/'|-∘-|'
¯3 6 6 1 1 1 1 2                |----∘----|

      1 0 0 1/¯3 6 1 2              1 0 1 0 0/'|-∘-|'
¯3 2                            |∘
```

If the left argument is Boolean, replicate is called the *compression* function; the vector is compressed by omission of terms corresponding to the zeros in the left argument:

```
      X                          (X>3)/X                (X≠3)/X
3 2 4 7 3 4 5 9 3   4 7 4 5 9              2 4 7 4 5 9
```

7.10 Example: Separating Data

The 60-element vector *C* below contains a mixture of data from two distinct populations. A histogram of the data makes that fact apparent. Looking at the ordered data, *C*[⍋*C*], it is clear that the gap in the data is about 32. Notice the use of (*C*<32)/*C* to select the data in *C* that is less than 32:

```
      C
54 21 25 24 45 22 53 47 29 39 42 47 49 41 45 30 25 44 23 24 27
      37 24 21 47 47 45 39 22 49 18 51 23 43 21 41 43 23 42 24
      25 24 26 22 49 24 35 53 36 25 23 27 41 44 18 22 45 20 46
      24
      C[⍋C]
18 18 20 21 21 21 22 22 22 22 23 23 23 23 24 24 24 24 24 24 24
      25 25 25 25 26 27 27 29 30 35 36 37 39 39 41 41 41 42 42
      43 43 44 44 45 45 45 45 46 47 47 47 47 49 49 49 51 53 53
      54


      HIST C
      *
      *
      *
     ****                    * *
    *****                 *    * * *
*   ***** *             * ****** * *    *
* ******** **    *** * ******** * * **
```

```
      C1←(C<32)/C                    C2←(C>32)/C
      C1                             C2
21 25 24 22 29 30 25 23 24       54 45 53 47 39 42 47 49 41
      27 24 21 22 18 23 21             45 44 37 47 47 45 39
      23 24 25 24 26 22 24             49 51 43 41 43 42 49
      25 23 27 18 22 20 24             35 53 36 41 44 45 46


      AVG C1                         AVG C2
23.53333333                    44.63333333
```

```
      HIST C1
        *
        *
        *
      ****
     *****
*    *****  *
* ********* **
```

```
      HIST C2
           * *
         *   * * *
       * ***** * *    *
*** * ******** * * **
```

7.11 Scan

The *scan operator,* designated with a backslash, \, is used to compute partial reductions, such as a vector of partial sums or partial products of a given vector. The expression +\2 5 6 4 results in the vector 2 7 13 17; that is, the vector of partial sums (+/2), (+/2 5), (+/2 5 6), (+/2 5 6 4). Notice that ***when a reduction applies to a single element, the result is the element.*** Likewise ×\2 5 6 4 results in the vector of partial products 2 10 60 240. Scan extends to arrays in much the same way that reduction does; examples are

```
      A
1  2  3  4
5  6  7  8

      +/A
10 26

      +\A
1  3  6 10
5 11 18 26

      +⍀A
1  2  3  4
6  8 10 12

      ×\A
1  2   6   24
5 30 210 1680
```

```
      B
0 0 1 1 1
0 1 1 1 1

      +/B
3 4

      +\B
0 0 1 2 3
0 1 2 3 4

      +⍀B
0 0 1 1 1
0 1 2 2 2

      <\B
0 0 1 0 0
0 1 0 0 0
```

First a reduction.

Notice that +\A results in partial sums along the rows of A, and +⍀A results in partial sums along the columns of A. If *f* is any primitive scalar dyadic function and A is an array, then *f*\A gives the partial *f* reductions along the last axis of A. Of course, for some scalar dyadic functions, the domain may be restricted. The scan operator is used next in a variation on the histogram function.

7.12 Example: Cumulative Histogram

The function HIST is modified to show the accumulation of data. The only change to HIST is the use of scan on line 2 so that the cumulative frequencies, or the scan of the frequencies, SF, are

computed instead of the frequencies *F* that were used in *HIST*:

```
     ∇ Z←CUMHIST W;M;A;SF
[1]    A←M+¯1+⍳1+(⌈/W)-M←⌊/W
[2]    SF←+\+/A∘.=W
[3]    Z←' *'[1+(M+1-⍳M←⌈/SF)∘.≤SF]
     ∇

     D←3 5 4 3 3 3 7 8 5 2 2 1
     HIST D
  *
  *
 ** *
******  **
     CUMHIST D
       *
      **
     ****
     ****
    *****
   ******
   ******
   ******
   ******
  *******
  *******
 ********
```

7.13 Plotting X–Y Points

Consider the problem of plotting a given list of (X, Y) points by marking a spot corresponding to each (X, Y) with a star. For simplicity, assume the points have integer entries. Each column of a 2-by-n matrix will be used to represent a point. The matrix *AXY* given below represents the list of points (1,4), (2,4), (4,3), (1,1), (5,2).

```
      AXY
1  2  4  1  5
4  4  3  1  2
```

A direct method of approaching this problem would be to construct an appropriate blank character matrix and to change the characters corresponding to each point into a star one at a time. This suggests a loop, since the number of points presumably can vary. A different approach is presented here.

In the following example, the stars in the positions required to plot the points given in *AXY* are indicated on the left. The points in *AXY* are underlined in the lattice of (X, Y) indices given in the middle. On the right, single indices corresponding to the (X, Y) double indices are underlined:

```
Y
4 | * *          (1,4) (2,4) (3,4) (4,4) (5,4)
3 |     *        (1,3) (2,3) (3,3) (4,3) (5,3)
2 |       *      (1,2) (2,2) (3,2) (4,2) (5,2)
1 | *            (1,1) (2,1) (3,1) (4,1) (5,1)
  +-------
    1 2 3 4 5 X
```

+	1	2	3	4	5
0	1	2	3	4	5
5	6	7	8	9	10
10	11	12	13	14	15
15	16	17	18	19	20

The correspondence between the double indices and the single indices is used by the plotting function for impressive efficiency.

First notice that the table of single indices is given by `0 5 10 15 ∘.+ 1 2 3 4 5`. That is, the single index corresponding to (X, Y) is given by the X value added to 5 times 1 less than the row number. But 1 less than the row number is just $4-Y$. Therefore, the single index is given by $X+5\times 4-Y$. Thus, `1` corresponds to $(1,4)$ since `1+5×4-4` is `1`, and `15` corresponds to $(5,2)$ since `5+5×4-2` is `15`. In general, if X is bounded by `1` and R, and Y is bounded by `1` and S, the single index corresponding to the point (X, Y) is given by $X+R\times S-Y$.

The function *PLOTXY* given below takes a list of points given as a two-row matrix and results in a character matrix plotting the points. The minimum X value is subtracted from the vector of X values, and one more is added so that X has minimum element `1`; this convention saves space in the result. The Y data are adjusted similarly. A blank vector is created on line 3, and the positions given by the indices $X+R\times S-Y$ are assigned stars on line 4. Line 5 reshapes the vector into the required matrix:

```
      ∇ Z←PLOTXY XY;R;S;X;Y
[1]     R←⌈/X←1+XY[1;]-⌊/XY[1;]      Get X, R.
[2]     S←⌈/Y←1+XY[2;]-⌊/XY[2;]      Get Y, S.
[3]     Z←(S×R)ρ' '                  Let Z be a blank vector.
[4]     Z[X+R×S-Y]←'*'               Change blanks at X+R×S-Y to '*'.
[5]     Z←(S,R)ρZ                    Reshape Z into a matrix.
      ∇
      AXY
1 2 4 1 5
4 4 3 1 2
      PLOTXY AXY
**
  *
   *
*

      P
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
 8  6  5  6  4  3  3  4  4  4  2  3  4  1  1
      PLOTXY P
*

 * *
  *
    *   ***   *
     **     *
          *
              **
```

7.14 *Example: Least-Squares Line Fitting*

In this section the line that best fits given data will be computed. The least-squares system solver of APL is introduced here to find that line. Finding the least-squares solution to a linear system was considered in greater generality in Chapter 4.

First consider the problem of finding the line $y = b + mx$ through two points — for example, through the points $(2, 5)$ and $(4, 3)$. This requires that b, m be found so that $b + 2m = 5$ and $b + 4m = 3$. This system can be written in $AX = C$ matrix form as follows:

$$\begin{bmatrix} 1 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} b \\ m \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \end{bmatrix}, \quad \text{where } A = \begin{bmatrix} 1 & 2 \\ 1 & 4 \end{bmatrix}, \quad X = \begin{bmatrix} b \\ m \end{bmatrix}, \quad C = \begin{bmatrix} 5 \\ 3 \end{bmatrix}$$

In APL the solution, *X*, to the system $AX = C$ is given by *C*⌹*A*. The symbol ⌹ is called quad-divide or domino and is used dyadically to invoke the APL *system solver* function. For this example the matrix *A* and vector *C* are entered and the vector *X* that gives the solution is computed:

```
      A←2 4 ∘.* 0 1
      C←5 3
      A
1 2
1 4
      X←C⌹A
      X
7 ¯1
```

Thus the line through the points $(2, 5)$ and $(4, 3)$ is $y = 7 - x$.

Next consider the problem of finding a line through several points. Typically, it is impossible to find a line through many given points. The problem now is to find the line of least-squares fit to the points. The same APL symbol domino as in *C*⌹*A* computes the least-squares solution to a linear system $AX = C$.

For example, in order to find the coefficients *L* of the line of least-squares fit to the data below, the expression *L*←*D*[2;]⌹*D*[1;]∘.* 0 1 is used. The data *D* give the tire width in millimeters and the mileage in miles per gallon for several makes of cars:

```
      D
185 185 175 185 215 195 165 205 205 205 185 175 195
 32  28  31  29  27  27  34  29  30  24  31  36  31

      PLOTXY D
          *


*
                    *
          *         *           *
                                      *
                    *                 *
                    *
                                *                 *


                                      *

      L←D[2;]⌹D[1;]∘.* 0 1
      L
59.69969512 ¯0.156402439
```

In order to plot the data from *D* and the line of best fit whose coefficients are in *L*, the function *PLOTXYC* from Exercise 17 will be considered. *PLOTXYC* takes a three-row matrix as its argument. The first two rows give the (X, Y) points as were required for *PLOTXY*. The third row gives a positive integer that determines which of several different markers is to be used to plot the (X, Y) point given in the same column. For example,

```
      T
1 2 3 4 5 6 1 2 3 4 5 6 6
1 2 3 4 5 6 6 5 4 3 2 1 3
1 1 1 1 1 1 2 2 2 2 2 2 3

      PLOTXYC T
o     *
 o   *
  o *
  * o +
 *   o
*     o
```

Notice that the diagonal points $(1, 1), (2, 2), \ldots, (6, 6)$ are plotted with the first marker, which is a star. The points on the opposite diagonal $(1, 6), (2, 5), \ldots, (6, 1)$ are plotted with the second marker, which is a jot. The single point $(6, 3)$ is plotted with the third marker, which is a plus sign.

Reconsider the tire width and mileage data *D* and the line of best fit to those data that have coefficients given in *L*. Once the coefficients of the line are known, the *Y* values of the best fit line can be computed for the integer *X* values between 165 and 215. The matrix *FIT* below contains the *X* data in the first row and the rounded *Y* values in the second row. In order to plot the points in *D* with the first marker, the function *PLOTXYC* could be applied to the matrix *D* augmented with a bottom row of 1s: *D*,[1]1. In order to plot the points in *FIT* with the second marker, the function *PLOTXYC* could be applied to the argument *FIT*,[1]2. Below, *PLOTXYC* is applied to (*D*,[1]1),*FIT*,[1]2 so that the original *D* data are plotted with the first marker, a star, and the points in *FIT* are plotted with the second marker, a jot:

```
X←164+⍳51                    Integers between 165 and 215.
Y←(X∘.*0 1)+.×L              The Y values on the best fit line.
FIT←2 51⍴X,⌊.5+Y             Rounded points on the best fit line.

PLOTXYC (D,[1]1),FIT,[1]2
      *


o o o
    o o o o o o
          o o o o o o o      *
           *       o o o o o o          *
                      o o o o o o o      *
                      *           o o o o o o      *
                      *                    o o o o o o
                                 *              o o o o o o o   *
                                                        o o o
                                                        *
```

The character graphics used here give a crude plot of the best fit line, which is marked with the jots. Many versions of APL provide interfaces with high-resolution graphics devices that would allow much finer plots.

7.15 Sampling

Sampling from binomial, exponential, and normal populations is considered below. Each of these sampling problems is handled in different ways, but all use random selection of arrays of real numbers from the interval [0, 1]. The function ***RANREAL*** from Section 3.4 provides the required random numbers. Its right argument specifies the shape of the required array:

```
     ∇ Z←RANREAL R
[1]    Z←1E¯17×?Rρ1E17
     ∇

       RANREAL 4
0.8863190227 0.5946735862 0.2477868957 0.001728506318

       RANREAL 2 2
0.2457823568 0.3120417967
0.1431395057 0.4182674714
```

7.16 Example: Sampling from an Exponential Population

The inverse probability distribution method allows sampling from the exponential density. That is, you apply the inverse of the cumulative distribution function to a uniform random variable (see Lewis and Orav). The exponential density has the explicit cumulative distribution $y = F(x) = 1 - e^{-\lambda x}$. Solving for x, you get $x = (-1/\lambda)\ \ln(1 - y)$. Given a random real, y, in [0, 1], choose the corresponding x from the cumulative distribution to get a random selection from an exponential population. Notice that if y is randomly chosen in [0, 1], then so is $1 - y$. The function ***SAMPEXP*** takes the number of repetitions ***R*** as its right argument and the population statistic λ, called ***LAM***, as its left argument. The data are spread by a factor of 40 before plotting to get a more informative histogram:

```
     ∇ Z←LAM SAMPEXP R
[1]    Z←(-÷LAM)×⍟RANREAL R
     ∇

       X←3 SAMPEXP 25
       AVG X
0.3359572623
       SD X
0.3782190582
       HIST ⌊40×X
  *
  *
* * *      *      *
* * * * *  * *  * *  *    *  *          * *            *  *             *
```

7.17 Example: Sampling from a Binomial Distribution

Binomial distributions arise by counting the number of successes in N trials of an experiment with probability of success P on each trial. This is simulated by comparing a random real in [0, 1] to P; if the random real is less than or equal to P, then the trial is considered a success.

The function *SAMPBIN* simulates the experiments for R repetitions of N trials using an intermediate R-by-N Boolean matrix with elements that are the result of each individual trial. Summing the rows of the matrix counts the number of successes in N trials as required. The left argument of *SAMPBIN* is *NP*, which is a two-element vector. The entry *NP*[1] gives the number N of trials of each experiment, and *NP*[2] gives the probability P of success in each trial:

```
     ∇ Z←NP SAMPBIN R
[1]    Z←+/NP[2]≥RANREAL R,NP[1]
     ∇

       X←5 .45 SAMPBIN 40
       X
2 1 1 3 3 2 2 2 2 3 2 1 3 2 2 2 2 3 3 2 3 3 2 1 3 2 1 3 1 2 1 3
       3 2 2 3 1 4 1 0

       HIST X                AVG X                SD X
  *                    2.1                0.8711913446
  *
  *
  **
  **
  **
  **
 ***
 ***
 ***
 ***
 ***
 ***
 ***
 ***
*****
```

7.18 Example: Sampling from a Normal Population

For normal populations, a third approach to sampling will be used. Rather than working back from a cumulative distribution or simulating experiments as in the previous two examples, the approach will be to approximate a normal population by averaging many selections from a uniform density. This is slow and only approximate. See Exercise 31 or Lewis and Orav for a better method.

A short aside describes the arithmetic used. First, $N \leftarrow 25$ selections from a uniform distribution on [0, 1] are summed; then, $-N \div 2$ is added to get an expected value of 0. This is scaled by $\div N$ to get

an average of uniform random variables. The uniform density has standard deviation $\sigma = 1/\sqrt{12}$, and the average of N of these has standard deviation $\sigma/\sqrt{N}$. Thus, an additional scaling factor of `(12×N)*.5` is required to get a standard deviation of 1. Those factors are combined into a single factor of `(12÷N)*.5` in *SAMPNOR*. Lastly, multiplying by the desired standard deviation and adding the desired mean changes that approximately standard normal random variable into one with the desired characteristics.

The left argument of *SAMPNOR* is a two-element vector that gives the desired mean and standard deviation. The right argument is the number of repetitions required:

```
     ∇ Z←MS SAMPNOR R;N
[1]    Z←MS[1]+(MS[2]×((12÷N)*.5))×(-N÷2)++/RANREAL R,N←25
     ∇

       X←0 1 SAMPNOR 50
       X[⍳5]                    Display a few entries of X.
¯1.27006884 1.06817433 ¯0.442250897 ¯0.400636199 0.629383956
       RANGE X
¯2.05546352 1.76279731
       AVG X
¯0.219465735
       SD X
0.85453993

       HIST ⌊10×X               Scale by a factor of 10.
                    *
                    *   **
                *   *  *  *  ***
*         *  **  ****  *  ****           *
**    *  **  **********  *****  **  **   *  *
```

Exercises

1. Find the values of the indicated expressions involving the vector `A←1 ¯4 5 ¯2` and the matrix `B←3 4⍴3 5 6 ¯3 0 9 ¯1 7 2 3`.

   (a) `3⌊A` (b) `0⌈A`
   (c) `⌈/A` (d) `⌈/B`
   (e) `⌈≠B` (f) `⌊≠B`
   (g) `⌈/⌈/B` (h) `⌈/⌊/B`
   (i) `⌊/⌈/B`

2. The matrix *S* has the scores for four students on five exams. Write an expression to find the average of the four highest scores for each student.

```
       S
78  83  80  65  86
89  91  87  75  90
72  68  72  68  77
52  65  71  70  73
```

3. Give an expression for finding the average of the entries in a vector *V* after the highest and lowest entries are dropped. Test the expression on *V*←4 5 8 6 6 7 8 6.
4. Find the values of the following expressions. Let *A*←1 ¯2 3 0, *B*←5 4 3 2, and *C*←5 0 ¯3 2 1 5 4 5.

   (a) ⍋*A* (b) *A*[⍋*A*]
   (c) ⍋*B* (d) *B*[⍋*B*]
   (e) ⍋*C* (f) *C*[⍋*C*]

5. Find the result of the following:

   (a) 6 3⍴'*AEIOU*' (b) '*',3 3⍴'12'
   (c) ⍴'*JK*','*' (d) '*ABCDEFGH*'[2 3⍴1 4 5 2 3 2]
   (e) ' *'[1+'*W*'='*WHY WILL WE*?']

6. Use APL to find the mean, median, standard deviation, range, and order up the data:
   13 56 24 12 35 27 41 22 31.
7. Give APL expressions to produce the following character arrays:

   (a)
   ```
   *∘∘∘∘
   **∘∘∘
   ***∘∘
   ****∘
   *****
   ```
   (b)
   ```
   *∘∘∘
   ∘*∘∘
   ∘∘*∘
   ∘∘∘*
   ```
   (c)
   ```
   *∘∘∘
   *∘∘∘
   *∘∘∘
   ****
   ```
   (d)
   ```
   +--===<<<<|||||
   ```

8. Give an APL expression for determining whether the character vector *SENT* contains all the letters of the alphabet, ignoring case.
9. Determine the result of the following expressions:

   (a) 2 3 4/1 2 3 (b) 0 1 2/1 2 3
   (c) 5/3 2 1 (d) (*A*<4)/*A*←2 2 3 0 6 5 1
   (e) (*A*<4)/*A*×2 (f) (*A*>4)/*A*×2

10. What is required for vectors *U* and *V* so that *U*/*V* makes sense? What is the shape of the result? In particular, does it depend on both *U* and *V*?
11. The data *V* given below arise by mixing together data from three very distinct populations. Make a reasonable separation of the populations and find the mean and range of each.

```
          V
40 67 65 25 66 41 21 45 43 42 63 43 69 75 46 71 44 43 59 76 43
      30 69 43 72 41 42 67 68 71 43 45 42 44 42 43 41 18 67 70
      43 62 26 68 44 39 40 44 44 65 42 64 16 69 70 72 13 21 69
      41 42 73
```

12. Write an APL function *MODE* that computes the mode(s) of a vector of integral data.
13. Write an APL function *HISTX* that provides a histogram of data with unnumbered axes added to the histogram. Include the origin, indicating it with a plus symbol. Have a message displayed that gives the range of the data. See Section 3.12.
14. Determine the result of the following expressions:

   (a) +\⍳10 (b) ×\⍳10 (c) -\⍳10
   (d) ⌈\2 0 ¯2 1 5 3 2 7 2 ¯3
   (e) ⌊\2 0 ¯2 1 5 3 2 7 2 ¯3

15. Determine the result of the following for *A*←2 3⍴2 3 1 0 1 4:

   (a) +*A* (b) ×*A* (c) +⌿*A* (d) ×⌿*A* (e) ⌈*A* (f) ⌊*A*

16. Give an APL expression for the vector of partial sums of the reciprocals of the factorials of the integers 0 to 9. Execute it.

17. Write an APL function *PLOTXYC* that plots points but allows several different markers to be used. The input should be a 3-by-N matrix. The first two entries of each column give the (X, Y) coordinates of a point, and the third element in the column is a digit 1 to 9 specifying which marker from among `' * o + O ▽ × # . '` should be used. (Hint: Modify the function *PLOTXY* in Section 7.13 in a manner suggested by the following example:
`W[2 3 6]←' * o + O ▽ × # . '[2 7 8]` for `W←10⍴' '`.)

18. Write an APL function *PLOTF I T* that does a scatter plot of data and also plots the least-square fit to the data. (Hint: Use Exercise 17.)

19. Write an APL function *SAMPUN I* that samples a uniform random variable on $[A, B]$ *R* times.

20. Write an APL function *PLOT2FCT* that plots two functions *FCT1* and *FCT2* on the same scale. Input the desired x interval and number of points at which the functions should be evaluated. Round the function values to the nearest integer. (Suggestion: Use *PLOTXYC* from Exercise 17 and the function *PART I T I ON* from Exercise 3.9.

21. **(a)** Write an APL function *CORF I T* that gives the y intercept and slope of the line of least-squares fit to a given list of (x, y) pairs of data; catenate to that the sample correlation coefficient for that fit.

(b) Use *CORF I T* on the data *D* in Example 7.14.

22. Write an APL function *SCH I ST* that produces a histogram scaled by a vertical factor of *M*. For example, if `M←10` it will plot two stars in a column corresponding to a frequency in the range 15 to 24.

23. Sample 100 elements from a binomial distribution with $n = 7$, $p = .2$. Compute the mean and standard deviation and display your results in a histogram using *SCH I ST* from Exercise 22.

24. Sample 100 elements from an exponential density with $\lambda = 2$. Compute the mean and standard deviation and display your results in a histogram using *SCH I ST* from Exercise 22.

25. Sample 100 elements from a normal population with $\mu = 100$, $\sigma = 12$. Compute the mean and standard deviation and display your results in a histogram.

26. Use the function *SCH I ST* from Exercise 22 to produce a histogram of 1000 elements sampled from a normal distribution with $\mu = 0$ and $\sigma = 10$. Use a vertical scale factor of 10. (Remark: It is possible that your workspace is too small to sample 1000 elements at once. In that case make several samples of smaller size and then combine them to get a sample size of 1000.)

27. Repeat 10 times the experiment of averaging a sample of 25 elements from a normal population with $\mu = 10$ and $\sigma = 2$. What is the mean and standard deviation of those averages?

28. **(a)** Write an APL function that samples *R* times from a population with density function:

$$f(x) = \frac{1}{4500}(30x - x^2) \qquad \text{for } 0 \le x \le 30$$

(Hints: A loop is appropriate. *SAMPUN I* from Exercise 19 may be used.)

(b) Sample 50 times from the population in (a) and display your results in a histogram.

29. **(a)** Sampling from discrete populations can be handled by computing a cumulative distribution and then selecting the item corresponding to the first element of the cumulative distribution greater than a random real from [0, 1]. Write an APL function that samples *R* times from a given discrete distribution. (Hint: You may want to use a loop and *RANREAL*.)

(b) Sample 50 times from the following distribution and plot your results with a histogram:

x	0	1	2	3	4	5
$f(x)$	.1	.2	.4	.1	.1	.1

30. **(a)** Write an APL function that evaluates the probabilities of a Poisson distribution $P(x, \lambda)$ for a vector of x arguments.

(b) Find by experimenting a value ***N*** so that the cumulative Poisson distribution from **1** to ***N*** with $\lambda = 5$ is **1** to within machine precision.

(c) Write an APL function that samples ***R*** times from a Poisson distribution with $\lambda = 5$. (Hint: Use Exercise 29(a).)

31. Sampling from normal distributions may be accomplished more accurately and efficiently by using the fact that a binormal distribution has an exact formula for the distribution for the radius ρ, where $\rho^2 = X_1^2 + X_2^2$ and the X_i are standard normal. In particular, $\rho = -2 \ln(U)$ gives random radii from a binormal distribution when U is uniform on the interval [0, 1]. Write a function ***SAMPNORM*** that produces ***R*** samples from a normal distribution with specified mean and standard deviation. (Hint: Compute a random angle, use the given random radius, and compute a random X_2 from those.)

8

More Array Manipulation

This chapter introduces a variety of APL functions for manipulating arrays. They are take, drop, ravel, replicate, expansion, reverse, rotate, dyadic transpose, and laminate. These functions are used for selecting subarrays out of an array, filling out arrays, or rearranging data within arrays. Examples that use these tools include polynomial addition, polynomial multiplication, matching a given distribution, and construction of banded matrices. The array manipulation tools introduced here will greatly enhance your ability to construct natural and general solutions to computational problems.

8.1 Take and Drop on Vectors

APL provides primitive functions *take* and *drop*, denoted ↑ and ↓, respectively, which are used for selecting data from the ends or corners of arrays. When V is a vector and N is a positive integer, $N \uparrow V$ takes the first N entries and results in a vector containing those entries. Likewise $N \downarrow V$ drops the first N entries and results in the remainder of the vector V. For example,

```
      5↑2 4 6 8 10 12 14 16 18
2 4 6 8 10

      5↓2 4 6 8 10 12 14 16 18
12 14 16 18

      2↑1 0 2 4 1
1 0

      2↓1 0 2 4 1
2 4 1
```

When the integer N is negative, the elements are selected or deleted from the end of the vector V rather than from the beginning. When the magnitude of N is larger than ρV, zeros or blanks are padded onto vectors to fill them out to the required length, N. This use of take is called *overtake*. Zeros

are used to fill out numeric vectors and blanks are used to fill out character vectors. Examples are

```
      ¯2↑3 4 5 6 7
6 7
      7↑3 4 5 6 7
3 4 5 6 7 0 0
      ¯7↑3 4 5 6 7
0 0 3 4 5 6 7
      ¯2↓3 4 5 6 7
3 4 5
      (7↑'**'),'*'
**     *
      ¯5↑'HI JOE'
I JOE
```

8.2 Example: Polynomial Addition

Let *A* and *B* be vectors of the coefficients of polynomials in increasing degree order. If the polynomials are the same degree, then ρ*A* is ρ*B* so that *A*+*B* is defined and represents the sum of the polynomials. If the polynomials have different degrees, then one of them needs to be padded with some zero terms. Overtake makes that easy.

Consider the example of adding $x - 3x^2$ to $3 + 4x - x^2 + x^3 + 2x^4$, say

```
      A←0 1 ¯3
      B←3 4 ¯1 1 2
      5↑A
0 1 ¯3 0 0
      B+5↑A
3 5 ¯4 1 2
```

The last result is the vector representing the sum of the polynomials as desired. The function *POLYADD* takes vectors representing polynomials as arguments and results in a vector representing the sum of the polynomials. *POLYADD* uses the maximum function from Section 7.2:

```
    ∇ Z←A POLYADD B;M
[1]   M←(ρA)⌈ρB          Get the maximum length.
[2]   Z←(M↑A)+M↑B
    ∇
      3 4 ¯1 1 2 POLYADD 0 1 ¯3
3 5 ¯4 1 2
      0 1 ¯3 POLYADD 3 4 ¯1 1 2
3 5 ¯4 1 2
      1 2 3 POLYADD 1 ¯2
2 0 3
      1 1 1 POLYADD 1 0 0 1
2 1 1 1
```

8.3 Take and Drop on Matrices

The take and drop functions can be used with a matrix right argument if the left argument is a two-element vector. The result is a "corner" of the matrix argument. Negative signs in the left argument indicate that selection is from the end of the corresponding axis. Notice there are several possibilities

for selecting negative signs; for example, when *A* is a matrix 2 ¯3↑*A* selects the elements from the first two rows and the last three columns of *A*:

```
      A
1 2 3 3 3 3
4 5 6 3 3 3
7 8 9 3 3 3
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
```

```
      3 3↑A
1 2 3
4 5 6
7 8 9
```

```
      3 ¯3↑A
3 3 3
3 3 3
3 3 3
```

```
      ¯3 3↑A
0 0 0
0 0 0
0 0 0
```

```
      ¯3 ¯3↑A
1 0 0
0 1 0
0 0 1
```

```
      2 8↑A
1 2 3 3 3 3 0 0
4 5 6 3 3 3 0 0
```

```
      3 3↓A
1 0 0
0 1 0
0 0 1
```

```
      0 3↓A
3 3 3
3 3 3
3 3 3
1 0 0
0 1 0
0 0 1
```

```
      ¯2 0↓A
1 2 3 3 3 3
4 5 6 3 3 3
7 8 9 3 3 3
0 0 0 1 0 0
```

```
      2 4↓A
3 3
0 0
1 0
0 1
```

Notice that the expression 2 8↑*A* is an overtake—two columns of 0s are padded onto the end. If the left argument of take has a 0 entry, the result is empty. For example, 3 0↑*A* is an empty matrix with shape 3 0.

8.4 Ravel

The monadic use of a comma provides a function that strings out all of the data in a given array as a vector. This function is called *ravel*. For example,

```
      R
1 0 2
1 1 1
      ρR
2 3
      ,R
1 0 2 1 1 1
      ρ,R
6
```

```
      S
2 4 5 6 8
      ρS
5
      ,S
2 4 5 6 8
      ρ,S
5
```

```
      T
5
      ρT
(Empty vector.)
      ,T
5
      ρ,T
1
```

Notice that applying ravel to a matrix results in a decrease in dimension, but applying ravel to a scalar results in an increase in dimension. One use of ravel is for changing a scalar index into a vector

index. In particular, this can be used to easily obtain a column matrix from a matrix; for example,

```
      Q
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15

     Q[;2]
2 7 12

     ρQ[;2]
3

     Q[;,2]
 2
 7
12

     ρQ[;,2]
3 1
```

8.5 *Replicate, Compression, and Expansion*

Replication was considered for vectors in Chapter 7 and is reviewed and extended here. The left argument gives the number of repetitions of the corresponding entries of the right argument to be placed in the result. In the case that the left argument is Boolean, this function is called compression:

```
      X
2 5 3 1 4 4

      1 3 1 2 0 0/X
2 5 5 5 3 1 1

      (X≤3)/X
2 3 1

      Y
1 2 3 4

      Y/Y
1 2 2 3 3 3 4 4 4 4

      1 0 1 0/Y
1 3
```

This function extends to having matrix right arguments by having each entry of the vector left argument specify the number of repetitions of the corresponding columns of the right argument. Using ≠ instead of / with matrices allows replication of rows instead of the columns. That is, / designates replication along the last axis, and ≠ designates replication along the first axis:

```
      A
1  2  3  4
5  6  7  8
9 10 11 12

      2 0 1 0/A
1  1  3
5  5  7
9  9 11

      (2 0 1)≠A
1  2  3  4
1  2  3  4
9 10 11 12

      B
2 0 1 1 2
3 2 1 0 1

      (2=B[1;])/B
2 2
3 1

      2 1≠B
2 0 1 1 2
2 0 1 1 2
3 2 1 0 1
```

Expansion is like compression in that its left argument is Boolean. It is designated by a backslash \. For vector right arguments, its left argument contains a 1 in any position where an entry from the right argument is used and a 0 in any position where a fill entry is inserted. The fill characters are 0s for numeric arrays and blanks for character arrays as was the case for overtake:

```
      X
2 3 4 4
      1 1 0 0 1 0 1\X
2 3 0 0 4 0 4
```

```
        1 0 1 1 1\X
2 0 3 4 4
      0 1 1 0 0 1 1 0\X
0 2 3 0 0 4 4 0
```

Expansion, \, may also be used with matrices as a right argument and with the left argument being a Boolean vector indicating where to insert columns. Columns of 0s or blanks are inserted in the positions corresponding to a 0 of the left argument. The slash overstruck with a minus sign, ⍀, is used to expand so that rows are inserted. That is, \ is used to designate expansion along the last axis, and ⍀ designates expansion along the first axis; for example,

```
      C
1 2 3 4
5 6 7 8
```

```
        1 1 0 1 0 1\C
1 2 0 3 0 4
5 6 0 7 0 8
```

```
        1 0 0 1⍀C
1 2 3 4
0 0 0 0
0 0 0 0
5 6 7 8
```

```
      D
ABCD
EFGH
```

```
        1 1 0 1 0 1\D
AB  C D
EF  G H
```

```
        1 0 0 1⍀D
ABCD


EFGH
```

8.6 *Example: Constructing Data with a Given Distribution*

Consider the problem of producing a data set having a given distribution. For example, suppose you would like to construct a data set, *D*, with the following bimodal distribution. Unlike other sections in this chapter, this section assumes you have read Section 7.8 about *HIST* and Section 6.1 about deal:

```
      HIST D
*
***                          *
***                       ****
******* *             *********
******************************  **
```

The vector of desired frequencies, *FREQ*, can be determined by counting the number of asterisks in each column. Then replicate can be used to produce a data set that has the correct distribution.

Lastly, deal can be used to rearrange the data into random order:

```
      FREQ←5 4 4 2 2 2 1 2 1 1 1 1 1 1 1 1 2 2 2 2 3 3 3 4 2 1 1
1 0 1 1
      C←FREQ/⍳⍴FREQ
      C
1 1 1 1 1 2 2 2 2 3 3 3 3 4 4 5 5 6 6 7 8 8 9 10 11 12 13 14 15
      16 17 17 18 18 19 19 20 20 21 21 21 22 22 22 23 23 23 24
      24 24 24 25 25 26 27 28 30 31

      D←C[(⍴C)?⍴C]      Randomize indices using deal.
      D
24 4 17 12 22 31 5 25 1 28 10 2 23 26 8 4 22 27 20 9 3 5 1 2 25
      16 21 6 2 19 2 7 11 14 18 18 19 1 3 23 22 1 24 8 13 24 1
      21 24 15 30 3 23 20 21 17 6 3

      HIST D
*
***                        *
***                      ****
******* *            **********
******************************  **
```

8.7 Reverse

The monadic APL function *reverse*, denoted ⌽, is used to reverse the order of the entries along the last axis of an array. The symbol ⊖ is used to indicate the reversal is along the first axis:

```
      V                ⌽V               ⊖V
1 2 3 4          4 3 2 1          4 3 2 1

      A                ⌽A               ⊖A
1 2 3 4          4 3 2 1          5 6 7 8
5 6 7 8          8 7 6 5          1 2 3 4
```

For a vector V the first and the last axes are the same, hence ⌽V is the same as ⊖V. For a matrix A, however, ⌽A reverses the order of the columns and ⊖A reverses the order of the rows.

A vector is a palindrome if it is the same after its entries are reversed. Examples of palindromes are 'MADAM', 'OTTO', and 1 4 6 4 1. An easy test for whether a vector is a palindrome uses ⌽ from this section and ∧ from Chapter 5. A vector V is a palindrome if and only if ∧/V=⌽V yields a 1. The inner product V∧.=⌽V is an equivalent test.

8.8 Rotate

The data in arrays can be rotated into new positions with the dyadic function *rotate*. When V is a vector and N is an integer, the entries of V are rotated N positions by N⌽V, which is read N rotate V.

The entries are moved off the front and onto the back of the vector when N is positive and vice versa when N is negative:

```
      V                2⌽V               4⌽V
2 3 5 7 11       5 7 11 2 3        11 2 3 5 7

     ¯1⌽V              0⌽V               6⌽V
11 2 3 5 7       2 3 5 7 11        3 5 7 11 2
```

The row entries of a matrix A may be rotated by using $N⌽A$, where N is a vector with length matching the number of rows. The entries of N give the number of positions the corresponding row of A is rotated. In a similar manner, $N⊖A$ can be used to rotate column entries. Notice that when the left argument is a scalar it is extended to be a vector of conforming length:

```
      A                    2 1 3⌽A
1  2  3  4  0  0          3  4  0  0  1  2
5  6  7  8  0  0          6  7  8  0  0  5
9 10 11 12  0  0         12  0  0  9 10 11

      0 ¯1 ¯2⌽A              2⌽A
1  2  3  4  0  0          3  4  0  0  1  2
0  5  6  7  8  0          7  8  0  0  5  6
0  0  9 10 11 12         11 12  0  0  9 10

      1 0 1 0 1 0⊖A          1⊖A
5  2  7  4  0  0          5  6  7  8  0  0
9  6 11  8  0  0          9 10 11 12  0  0
1 10  3 12  0  0          1  2  3  4  0  0
```

8.9 *Example: Polynomial Multiplication*

Here the problem of multiplying two polynomials is considered. You need to compute all of the pairwise products of the monomial terms; this may be organized into a table. For example, to evaluate the product $(3 + 2x + x^2)(2 + x - 5x^2 + 3x^3)$, the following table could be computed:

	2	x	$-5x^2$	$3x^3$
3	6	$3x$	$-15x^2$	$9x^3$
$2x$	$4x$	$2x^2$	$-10x^3$	$6x^4$
x^2	$2x^2$	x^3	$-5x^4$	$3x^5$

Notice that like powers occur on the ascending diagonals. Hence, summing along those diagonals yields $6 + 7x - 11x^2 + x^4 + 3x^5$ as the final answer.

In APL the solution can be accomplished by first creating a table of products:

```
      U←3 2 1          Coefficients of one polynomial.
      V←2 1 ¯5 3       Coefficients of the other.
      U∘.×V            Make a times table.
6  3 ¯15  9
4  2 ¯10  6
2  1  ¯5  3
```

To effect the summation along the diagonals, entries are rotated to line up the coefficients of like powers in columns. This requires some 0 padding so that high-degree terms are not rotated around to columns of low-degree terms:

```
      U∘.×V,0×1↓U                          Some zero padding.
6  3 ¯15  9  0  0
4  2 ¯10  6  0  0
2  1  ¯5  3  0  0
      (1-⍳⍴U)⌽U∘.×V,0×1↓U                  Rotate the rows.
6  3 ¯15   9  0  0                         Columns contain entries from
0  4   2 ¯10  6  0                         ascending diagonals.
0  0   2   1 ¯5  3
      +⌿(1-⍳⍴U)⌽U∘.×V,0×1↓U                Add like terms.
6 7 ¯11 0 1 3
```

This treatment of polynomial multiplication is appealing because it solves a commonly occurring problem, uses an intermediate higher dimensional array, and achieves a concise solution.

8.10 Dyadic Transpose

The monadic use of transpose on a matrix was considered in Section 4.11. The expression ⍉*A* interchanges the rows and columns of *A*. Transpose is also a dyadic function; dyadic transpose is a generalization of monadic transpose. The expression 2 1⍉*A* is equivalent to ⍉*A*; that is, the first and second indices of *A* are interchanged. The expression 1 2⍉*A* yields *A* since the first index remains in the first position and likewise for the second index. The expressions 1 1⍉*A* and 2 2⍉*A* result in the elements of *A* whose indices match (some systems accept only 1 1⍉*A*); that is, the diagonal entries of *A*:

```
      A
 2  3  5  7
11 13 17 19

      1 2⍉A
 2  3  5  7
11 13 17 19

      ⍉A
2 11
3 13
5 17
7 19

      1 1⍉A
2 13

      2 1⍉A
2 11
3 13
5 17
7 19

      2 2⍉A
2 13
```

A comparison with standard mathematical notation is helpful for understanding the dyadic transpose. *R* is used to designate the vector or matrix results of the transposes of the matrix *A*:

APL Notation	Standard Mathematical Notation	
R←1 2⍉*A*	$R_{ij} = A_{ij}$	Where each index runs over the maximum meaningful range.
R←2 1⍉*A*	$R_{ij} = A_{ji}$	
R←1 1⍉*A*	$R_{i} = A_{ii}$	
R←2 2⍉*A*	$R_{j} = A_{jj}$	

8.11 Transpose on Higher Dimensional Arrays

For higher dimensional arrays, there are many ways of selecting the list of indices used with dyadic transpose. Notice that the monadic transpose reverses the order of the indices. Consider

```
A←2 3 4⍴⍳24:

      A
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

      ⍉A
 1 13
 5 17
 9 21

 2 14
 6 18
10 22

 3 15
 7 19
11 23

 4 16
 8 20
12 24

      3 2 1⍉A
 1 13
 5 17
 9 21

 2 14
 6 18
10 22

 3 15
 7 19
11 23

 4 16
 8 20
12 24

      1 3 2⍉A
 1  5  9
 2  6 10
 3  7 11
 4  8 12

13 17 21
14 18 22
15 19 23
16 20 24

      2 1 3⍉A
 1  2  3  4
13 14 15 16

 5  6  7  8
17 18 19 20

 9 10 11 12
21 22 23 24

      1 1 1⍉A
1 18

      1 2 2⍉A
 1  6 11
13 18 23

      2 1 1⍉A
 1 13
 6 18
11 23

      1 1 2⍉A
 1  2  3  4
17 18 19 20

      1 2 1⍉A
 1  5  9
14 18 22
```

A comparison with standard mathematical notation is again helpful for understanding dyadic transpose.

APL Notation	Standard Mathematical Notation	
`R←3 2 1⍉A`	$R_{ijk} = A_{kji}$	Where each index runs over the maximum meaningful range.
`R←1 3 2⍉A`	$R_{ijk} = A_{ikj}$	
`R←1 1 1⍉A`	$R_i = A_{iii}$	
`R←1 2 2⍉A`	$R_{ij} = A_{ijj}$	

8.12 *Laminate*

Laminate is an APL function that takes two arrays with the same shape and attaches them along a new axis of length 2. The new axis is indicated with a fractional axis index. The result is an array of dimension one more than its arguments.

If *U* and *V* are vectors of length *N*, then *U* , [.5] *V* is the 2-by-*N* matrix with rows *U* and *V* and *U* , [1 .5] *V* is the *N*-by-2 matrix with columns *U* and *V*. Notice that if one argument is a scalar, the scalar argument is extended to a vector of conforming length:

```
      U
1 2 3 4

      V
2 3 5 7

      V,[.5]8
2 3 5 7
8 8 8 8

      U,[.5]V
1 2 3 4
2 3 5 7

      U,[1.5]V
1 2
2 3
3 5
4 7

      V,[1.5]8
2 8
3 8
5 8
7 8
```

Recall that since *U* and *V* have only one axis, *U* , [1] *V* is catenation along that axis, yielding 1 2 3 4 2 3 5 7. Notice .5 is before 1, and *U* , [.5] *V* laminates *U* to *V* so that the new axis of length 2 is before the first and only axis of *U* and *V*. Likewise 1 .5 is after 1, and *U* , [1 .5] *V* puts the new axis of length 2 after the old first axis.

For matrices there are more possibilities. If *A* and *B* are matrices of the same shape, they may be laminated along a new axis before, in between, or after the two existing axes:

```
      A
1  2  3  4
5  6  7  8
9 10 11 12

      B
2 3 2 3
3 2 3 2
2 3 2 3

      ρA
3 4

      ρA,[.5]B
2 3 4

      ρA,[1.5]B
3 2 4

      ρA,[2.5]B
3 4 2

      A,[.5]B
1  2  3  4
5  6  7  8
9 10 11 12

2  3  2  3
3  2  3  2
2  3  2  3

      A,[1.5]B
1  2  3  4
2  3  2  3

5  6  7  8
3  2  3  2

9 10 11 12
2  3  2  3

      A,[2.5]B
 1  2
 2  3
 3  2
 4  3

 5  3
 6  2
 7  3
 8  2

 9  2
10  3
11  2
12  3
```

Higher dimensional arrays may also be laminated along a new axis before, in between, or after the existing axes.

8.13 Example: Constructing Banded Matrices

Diagonal and banded matrices occur frequently. They are used, for example, in computing splines and in many numerical methods for solving partial differential equations. Although it is possible to construct these with the primitive functions introduced in Chapter 2, it is convenient to construct them using laminate, replicate, rotation, drop, and take.

First consider the problem of constructing an N-by-N diagonal matrix with diagonal entries given in a vector V of length N:

```
      V
2 3 5 7
      V,[1.5]0          1 3/V,[1.5]0          (1-⍳4)⌽1 3/V,[1.5]0
2 0                 2 0 0 0               2 0 0 0
3 0                 3 0 0 0               0 3 0 0
5 0                 5 0 0 0               0 0 5 0
7 0                 7 0 0 0               0 0 0 7
```

Next consider the problem of constructing a banded matrix — that is, a matrix with many descending diagonals composed entirely of 0s. Here a 5-by-5 matrix C with 3s on the diagonal, 1s on the superdiagonal and subdiagonal, and 0s elsewhere is constructed:

```
      A←5 6⍴6↑1 3 1          B←(1-⍳5)⌽A          C←0 1↓B
      A                      B                   C
1 3 1 0 0 0           1 3 1 0 0 0           3 1 0 0 0
1 3 1 0 0 0           0 1 3 1 0 0           1 3 1 0 0
1 3 1 0 0 0           0 0 1 3 1 0           0 1 3 1 0
1 3 1 0 0 0           0 0 0 1 3 1           0 0 1 3 1
1 3 1 0 0 0           1 0 0 0 1 3           0 0 0 1 3
```

Exercises

1. Consider the vector `W←1 1 2 3 5 8`. Determine the result of the following APL expressions.

   (a) `4↑W` (b) `4↓W`
   (c) `¯4↑W` (d) `¯4↓W`
   (e) `8↑W` (f) `¯8↑W`
   (g) `0↑W` (h) `0↓W`
   (i) `8↓W` (j) `¯8↓W`

2. (a) Give conditions on the arrays A and B so that $A \uparrow B$ makes sense.
   (b) What is the shape of $A \uparrow B$?
   (c) Give conditions on $A \downarrow B$ so that it makes sense.
   (d) What is the shape of $A \downarrow B$?

3. Let `A←5 6⍴0 1 2 3 4`. Determine the result of the following expressions:

   (a) `3 4↑A` (b) `3 ¯4↑A`
   (c) `¯3 4↑A` (d) `¯3 ¯4↑A`
   (e) `3 4↓A` (f) `3 ¯4↓A`
   (g) `¯3 4↓A` (h) `¯3 ¯4↓A`
   (i) `0 4↓A` (j) `0 ¯4↓A`

4. Suppose a polynomial in two variables is represented by a matrix so that the row corresponds to the power of x and the column corresponds to the power of y:

```
      A
1  0  ¯5
2  3   8
```

	1	y	y^2
1	1	0	¯5
x	2	3	8

   Thus, `A` represents $1 + {}^{-}5y^2 + 2x + 3xy + 8xy^2$.

   (a) Write an APL function that takes two polynomials in two variables represented by matrices and results in the sum of the polynomials. No assumptions about the degrees of the polynomials is to be made ahead of time. (Hint: See Section 8.3.)
   (b) Use the function from part (a) to add $1 + 3xy^2 + y^4$ and $7 + 3x + 4x^2 + 2xy^2 + 3x^2y^2$

5. Let `A←3 4⍴⍳12 ⋄ B←⍳7 ⋄ C←5`. Determine the result of the following APL expressions:

   (a) `,A` (b) `⍴,A`
   (c) `,B` (d) `⍴,B`
   (e) `,C` (f) `⍴,C`
   (g) `A[3;]` (h) `⍴A[3;]`
   (i) `A[,3;]` (j) `⍴A[,3;]`

6. Give an APL expression that gives the shape of `,A` for any array `A` in terms of `⍴A`.
7. Modify ***POLYADD*** given in Section 8.2 so that it works with 0 degree polynomials that have been input as scalars rather than length 1 vectors.
8. Let `W←1 1 2 3 5 8`. Determine the results of the following APL expressions:

   (a) `1 0 1 0 1 0/W` (b) `1 2 1 2 2 2/W` (c) `W/W`
   (d) `1 1 0 1 1 0 1 1\W` (e) `0 1 1 1 0 0 1 1 1 0\W`

9. Let `A←3 5⍴0 1 2 3`. Determine the results of the following APL expressions:

   (a) `1 2 1 2 1/W` (b) `1 0 0 0 1/A` (c) `1 2 3≠A`
   (d) `1 0 0 1 0 0 1 0 0 1 0 0 1\A` (e) `¯(13⍴1 0 0)\A`
   (f) `1 0 1 1≠A`

10. Consider replication and expansion for a matrix `A` and an array `U`. What conditions on `A` and `U` are needed so that the following make sense?

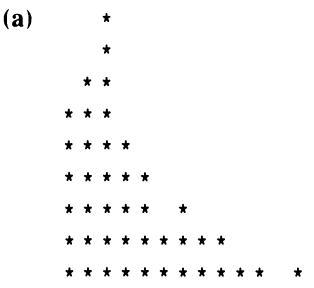
    (a) `U/A` (b) `U\A` (c) `U≠A` (d) `U⍀A`

11. Define an APL function ***EXPAND*** that expands a numeric matrix so that all the given entries are separated by a row and column of zeros. For example,

```
2 3 4 5                2 0 3 0 4 0 5
5 3 0 1     becomes    0 0 0 0 0 0 0
1 2 1 1                5 0 3 0 0 0 1
                       0 0 0 0 0 0 0
                       1 0 2 0 1 0 1
```

12. Construct data vectors that will have the following histograms. (Hint: See Section 8.6.)

(a)



(b)

13. **(a)** Suppose *A* is a Boolean matrix and the number of rows that are palindromes is to be computed. Give an APL expression for computing that number. (Hint: Use reverse.)

(b) Generate a random 200-by-10 Boolean matrix. How many rows are palindromes? Display the rows that are palindromes.

14. Write an APL function *POLYMULT* that computes the product of two polynomials. Make the arguments two vectors giving the coefficients of polynomials in increasing power order and the result a vector giving the coefficients of the product of those polynomials. See Example 8.9.

15. What happens if the function in Exercise 14 is run on vectors giving the coefficients in decreasing power order?

16. Let `W←1 1 2 3 5 8`. Determine the results of the following APL expressions:

(a) `2⌽W` **(b)** `¯2⌽W` **(c)** `⌽W` **(d)** `⊖W`

17. Let `A←3 5⍴0 1 2 3`. Determine the results of the following APL expressions:

(a) `0 1 2⌽A` **(b)** `0 1 2 3 4⊖A`
(c) `0 ¯1 ¯2⌽A` **(d)** `1⌽A`
(e) `⌽A` **(f)** `⊖A`

18. Consider rotate and reverse on a three-dimensional array. Let `A←2 3 4⍴⍳24`. Experiment with the following expressions. Try to predict the results of your experiments.

(a) `(2 3⍴1 2)⌽A` **(b)** `(2 4⍴1 2)⌽[2]A`
(c) `(3 4⍴1 2)⌽[1]A` **(d)** `⌽A`
(e) `⊖A` **(f)** `⌽[1]A`
(g) `⌽[2]A` **(h)** `⌽[3]A`

19. **(a)** Use your experience with Exercise 18(a) to give conditions on arbitrary arrays *R* and *A* so that *R*⌽*A* is defined.

(b) In that case, what is the shape of *R*⌽*A*?

20. Let `⎕RL←16807` and construct an array that gives a simulation of the number of hours spent each day grading linear algebra assignments for three students:

```
A←(2=?3 14 7⍴2)×?10 0 5∘.+14 7⍴5 2 4 2 4 2 1
```

Each column represents the day of the week, and the rows give the week of the 14-week semester. The three blocks give the hours worked by Angela, Sherrie, and Sharad, respectively. Experiment to find an APL expression that displays the data so that each block is a week, the columns are still days of the week, and the rows of each block correspond to the students.

21. Let `B←1 2 ∘.× 4 5 6 ∘.- 1 2 3 4`. Experiment with the following APL expressions. Try to predict the results of your experiments.

(a) `1 2 3⍉B` (b) `1 3 2⍉B`
(c) `3 2 1⍉B` (d) `3 1 2⍉B`
(e) `⍉B` (f) `1 1 2⍉B`
(g) `2 2 1⍉B` (h) `1 2 1⍉B`
(i) `1 1 1⍉B`

22. Write an APL function *DIAG* that takes a numeric vector argument *V* and results in a diagonal matrix with the given vector on the diagonal.
23. Suppose *A* and *B* have shape `5 6`. What are the shapes of the following?

(a) `A,B` (b) `A,[.5]B`
(c) `A,[1]B` (d) `A,[1.5]B`
(e) `A,[2]B` (f) `A,[2.5]B`

24. Suppose *A* and *B* have shape `5 6 7`. What are the shapes of the following?

(a) `A,B` (b) `A,[.5]B`
(c) `A,[1]B` (d) `A,[1.5]B`
(e) `A,[2]B` (f) `A,[2.5]B`
(g) `A,[3]B` (h) `A,[3.5]B`

25. Write an APL function, *TRIDIAG*, that produces an *N*-by-*N* tridiagonal matrix. It should take a three-element vector, *W*, as right argument, where `W[1]` gives the common subdiagonal entries, `W[2]` gives the common diagonal entries, and `W[3]` gives the common superdiagonal entries. The left argument should be the size *N*.

9

Sorting and Coding

This chapter considers the sorting of arrays according to several criteria. It treats alphabetization as a simple application of sorting once the character data are represented as positions in an alphabet. It introduces modular arithmetic and uses it to modify the alphabetizing scheme. Construction of finite field tables is straightforward with modular arithmetic, and the field of order 2 is interpreted in terms of logical functions. Data can be represented in other bases and reinterpreted as numbers. As an example, ordinary text is encoded as a binary string to which the simplest error-correcting Hamming code is applied.

9.1 Grade up, Grade Down

The APL *grade up* function, denoted ⍋, results in a list of indices. If X is a vector with distinct entries, then ⍋X is the vector containing the index of the smallest element of the vector X, followed by the index of the next smallest element of X, and so on, up to the index of the largest element of X. If the vector X has repeated entries, the indices of the repeated entries appear in increasing order. In Section 7.4 the APL idiom X[⍋X] was used to order the elements of the vector X. For example,

```
      U
6 2 3 0 9 2.4
      ⍋U
4 2 6 3 1 5
      U[⍋U]
0 2 2.4 3 6 9
```

```
      V
5 2.4 2.4 6 5
      ⍋V
2 3 1 5 4
      V[⍋V]
2.4 2.4 5 5 6
```

Likewise, the *grade down* function, denoted ⍒, results in a list of indices, with the indices of the larger elements first. If the vector has repeated entries, the indices of the repeated entries appear in increasing order:

```
      ⍒U
5 1 3 6 2 4
      U[⍒U]
9 6 3 2.4 2 0
```

```
      ⍒V
4 1 5 2 3
      V[⍒V]
6 5 5 2.4 2.4
```

Notice that U[⍋U] is ⌽U[⍒U] and likewise for V. However, ⍋U is ⌽⍒U, and ⍋V is not ⌽⍒V. In general if a vector V has repeated entries, then ⍋V and ⌽⍒V are different.

Iterating the grade up function produces the ordering ranks of the entries of the argument vector:

```
      U                    V
6 2 3 0 9 2.4        5 2.4 2.4 6 5
      ⍋U                   ⍋V
4 2 6 3 1 5          2 3 1 5 4
      ⍋⍋U                  ⍋⍋V
5 2 4 1 6 3          3 1 2 5 4
      ⍒U                   ⍒V
5 1 3 6 2 4          4 1 5 2 3
      ⍋⍒U                  ⍋⍒V
2 5 3 6 1 4          2 4 5 1 3
```

Notice that ⍋⍋V gives the rankings of the entries of V in increasing order, with ties decided by index order. In particular, the 6 in V is the fifth lowest entry of V. The rankings in decreasing order are given by ⍋⍒V.

9.2 Grade up, Grade Down on Matrices

Many versions of APL support monadic grade up on matrix arguments. Grade up applied to a matrix gives the indices of rows ordered lexicographically. Thus, if A is a matrix, ⍋A is a vector of row indices, where the first entry is the index of the row that is smallest in the following sense: A row A[I;] is smaller than A[J;] if in the first position in which they differ A[I;] is smaller or if the row A[I;] is identical to A[J;] but I is less than J. Grade down is similar. For example,

```
      A                    ⍋A                   A[⍋A;]
1 0 2 5              3 1 4 6 5 2          1 0 1 1
3 8 1 1                                   1 0 2 5
1 0 1 1                                   1 0 2 5
1 0 2 5                                   2 1 1 1
3 1 2 1                                   3 1 2 1
2 1 1 1                                   3 8 1 1

      A[⍋A[;1];]           A[⍋A[;3 4];]         A[⍒A;]
1 0 2 5              3 8 1 1              3 8 1 1
1 0 1 1              1 0 1 1              3 1 2 1
1 0 2 5              2 1 1 1              2 1 1 1
2 1 1 1              3 1 2 1              1 0 2 5
3 8 1 1              1 0 2 5              1 0 2 5
3 1 2 1              1 0 2 5              1 0 1 1

      A[⍋A[;1 3];]         A[⍋A[;3 1];]
1 0 1 1              1 0 1 1
1 0 2 5              2 1 1 1
1 0 2 5              3 8 1 1
2 1 1 1              1 0 2 5
3 8 1 1              1 0 2 5
3 1 2 1              3 1 2 1
```

Notice that `A[⍋A[;1];]` orders the rows of *A* by ordering the first column of *A* in the grade up sense. The other columns are ignored for purposes of determining the order. Likewise, `A[⍋A[;3 4];]` orders the rows of *A* so that the rows of the third and fourth columns are lexicographically ordered.

9.3 Example: Manipulation of Grade Point Average Statistics

Consider a table of the grade point averages *GPA*, of the Computer Science Club members. The first column gives the student number, the second column gives the faculty identification number of the student's advisor, the third column gives the semester GPA, and the last column gives the cumulative GPA. These can be ordered several ways. The expression `GPA[⍋GPA[;2];]` groups the information by advisor. The expression `GPA,⍋⍒GPA[;4 3]` gives the table plus a last column that gives the rankings of the students according to highest cumulative GPA with ties settled by looking at the semester GPA:

```
      GPA
4021  25  4    3.4
4034  19  2.1  1.8
4069  19  3.2  2.9
4080  19  1.6  2.2
4081  25  3.2  3.5
4087  18  3.5  2.9
4089  18  2.4  2.2

      GPA[⍋GPA[;2];]
4087  18  3.5  2.9
4089  18  2.4  2.2
4034  19  2.1  1.8
4069  19  3.2  2.9
4080  19  1.6  2.2
4021  25  4    3.4
4081  25  3.2  3.5

      GPA[⍒GPA[;3];]
4021  25  4    3.4
4087  18  3.5  2.9
4069  19  3.2  2.9
4081  25  3.2  3.5
4089  18  2.4  2.2
4034  19  2.1  1.8
4080  19  1.6  2.2

      GPA[⍒GPA[;4];]
4081  25  3.2  3.5
4021  25  4    3.4
4069  19  3.2  2.9
4087  18  3.5  2.9
4080  19  1.6  2.2
4089  18  2.4  2.2
4034  19  2.1  1.8

      GPA[⍒GPA[;4 3];]
4081  25  3.2  3.5
4021  25  4    3.4
4087  18  3.5  2.9
4069  19  3.2  2.9
4089  18  2.4  2.2
4080  19  1.6  2.2
4034  19  2.1  1.8

      GPA,⍋⍒GPA[;4 3]
4021  25  4    3.4  2
4034  19  2.1  1.8  7
4069  19  3.2  2.9  4
4080  19  1.6  2.2  6
4081  25  3.2  3.5  1
4087  18  3.5  2.9  3
4089  18  2.4  2.2  5
```

9.4 Index Relative to a Vector

The dyadic use of iota, `⍳`, provides a method of finding the indices of data in a reference vector. The left argument is the reference vector, and the right argument is data whose positions in the reference vector are desired. For each entry in the right argument, the index of the first occurrence of that ele-

ment in the left argument is given. If an entry in the right argument does not appear in the left argument, then an index 1 larger than the greatest meaningful index of the left argument is given for that entry. The expression $A \iota W$ is read "the A indices of W" or "the indices of W in A":

```
      8 5 7 9 ⍳ 5
2

      8 5 7 9 ⍳ 8 9 8
1 4 1

      'ABCDEFG' ⍳ 'BEG'
2 5 7

      'ABCDEFG' ⍳ 'LEG'
8 5 7

      A
2 7 7
5 9 5
```

```
      8 5 7 9[2]
5

      8 5 7 9[1 4 1]
8 9 8

      'ABCDEFG'[2 5 7]
BEG

      'ABCDEFG' ⍳ 'LOT'
8 8 8

      2 3 5 7 11 ⍳A
1 4 4
3 6 3
```

9.5 Example: Alphabetization

Grade up and index can be used to alphabetize a list of names. First dyadic iota is used to get a matrix of numbers from a matrix of letters. Grade up is then applied to the numeric matrix. Consider the 6-by-6 matrix of names N and the alphabet ALF. Notice that a blank is the first entry in ALF:

```
      ALF←' ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      N←6 6⍴'JOHN  BILL  THOMASJASON JACK  EDWARD'
```

```
      N                ALF⍳N                N[⍋ALF⍳N;]
JOHN          11 15  9 15  1  1      BILL
BILL           3 10 13 13  1  1      EDWARD
THOMAS        21  9 16 14  2 20      JACK
JASON         11  2 20 16 15  1      JASON
JACK          10  2  4 12  1  1      JOHN
EDWARD         6  5 24  2 19  5      THOMAS
```

9.6 Example: Removing Duplicates and Frequency Revisited

Dyadic iota yields interesting information when both arguments are the same vector. The resulting list of indices gives the indices of the first occurrences of any given element. These first occurrences are notable since their indices match the indices of index generator. Duplications in a vector X may be removed by selecting only the first occurrence of each element; namely, those elements of X that correspond to places where $X \iota X$ matches $\iota \rho X$.

Consider the vector X, which, for convenience, only contains integers; The same expressions work for real numeric vectors:

```
      X
8 4 2 2 8 7 4 4 6 1 2 5 7 7 2 4 7 1 7 1

      X⍳X
1 2 3 3 1 6 2 2 9 10 3 12 6 6 3 2 6 10 6 10

      ⍳⍴X
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

      (X⍳X)=⍳⍴X                    Note where indices from X⍳X match ⍳⍴X.
1 1 1 0 0 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0

      ((X⍳X)=⍳⍴X)/X
8 4 2 7 6 1 5
```

Thus $((X \iota X) = \iota \rho X) / X$ gives an expression that yields the elements of X with duplicates removed. This is called the *nub* of X. Notice that the elements in the nub of X appear in the order of their first appearance in X. Note, also, that the nub of an arbitrary array may be computed by applying the formula to the ravel of the array.

The elements of the nub of X can be ordered after the nub is computed if that is desired, but ordering X first suggests an efficient method for computing frequencies. Notice that if Y is X with elements put into ascending order, the frequency of a given element is apparent; see Y below. The frequencies of elements are the differences in the indices of first appearance of successive elements in Y. The expression $(Y \iota Y) = \iota \rho Y$ marks the positions where elements first appear, the nub is computed, and $((Y \iota Y) = \iota \rho Y) / \iota \rho Y$ lists the indices of those positions:

```
      Y←X[⍋X]                      Order the elements.
      Y
1 1 1 2 2 2 2 4 4 4 4 5 6 7 7 7 7 7 8 8

      (Y⍳Y)=⍳⍴Y                    Mark positions of new elements in Y.
1 0 0 1 0 0 0 1 0 0 0 1 1 1 0 0 0 0 1 0

      NUB←((Y⍳Y)=⍳⍴Y)/Y            The nub in order.
      NUB
1 2 4 5 6 7 8

      IN←((Y⍳Y)=⍳⍴Y)/⍳⍴Y           The indices of the first appearances.
      IN
1 4 8 12 13 14 19
```

The differences in those positions gives the frequency of the elements in the nub. Thus, there are 8 minus 4 appearances of 2. The value 8 appears twice; that is, 21 minus 19, where 21 is 1 more

than the number of elements in *Y*. Organizing the computation of those differences gives

```
      FREQ←(1↓IN,1+⍴Y)-IN
      FREQ
3 4 4 1 1 5 2

      NUB
1 2 4 5 6 7 8
```

Thus, there are three 1s, four 2s, four 4s, and so on, appearing in *Y*.

This method for computing frequencies is slightly more complicated than the method discussed in Section 7.7, but it is more efficient for large vectors. This is because simple vector operations are used instead of a table. The most time consuming operation used in this section is ordering a vector; however, that can be done very efficiently.

9.7 *Index Origin*

All of the APL functions considered so far have been presented under the assumption that counting begins with 1; thus ⍳3 is 1 2 3. In APL you can select counting to begin with 0 instead; with that selection ⍳3 is 0 1 2. This selection is made by assigning the value 0 to the system variable *index origin*, ⎕IO. The default value for ⎕IO in a clear workspace is 1. For example,

```
      ⎕IO                          Display the default value.
1
      ⍳5                 ?6⍴2                 'ABCDEFG'⍳'BAG'
1 2 3 4 5          2 1 2 2 1 1          2 1 7

      ⎕IO←0
      ⍳5                 ?6⍴2                 'ABCDEFG'⍳'BAG'
0 1 2 3 4          1 1 0 1 0 1          1 0 6
```

In general any APL function that uses indices is affected by ⎕IO. These include monadic and dyadic ⍳, roll (also known as random index generator), deal, indexing arrays, indexing axes (such as by catenate, laminate, transpose, rotate, and reverse), and grade up and grade down. Some more examples when ⎕IO is 0 follow:

```
      A                  A[0;]                A[1;]
1 2 3 4            1 2 3 4              5 6 7 8
5 6 7 8

      A[;2]              A,[0]A               1 0 ⌽A
3 7                1 2 3 4              1 5
                   5 6 7 8              2 6
                   1 2 3 4              3 7
                   5 6 7 8              4 8

      ⍉[0]A              A,[1]A               ⍴A
5 6 7 8            1 2 3 4 1 2 3 4      2 4
1 2 3 4            5 6 7 8 5 6 7 8
```

Note that shape and reshape are independent of ⎕IO. Grade up and grade down are sensitive to ⎕IO, whereas the idioms V[⍋V] and V[⍒V] for a vector V are not sensitive to ⎕IO. Again let ⎕IO be 0:

```
      V                ⍋V              V[⍋V]
5 2.4 2.4 6 5    1 2 0 4 3        2.4 2.4 5 5 6
```

Using index origin 0 has advantages in several mathematical and computer settings. In the case of polynomial computations, this advantage occurs since the constant terms are of degree 0. For character manipulation, it is useful because tables of ASCII characters are given starting with character 0. For modular arithmetic, it is natural because the standard residues modulo n include 0, as discussed below.

Functions that use different index origins may reside in the same workspace. It is possible and often wise to localize ⎕IO in function definitions. This prevents these different values from conflicting with each other or the global value of ⎕IO in the workspace.

9.8 Example: Polynomial Evaluation in Both Index Origins

Consider the problem of evaluating $P(t) = 3 + 2t - 5t^2 + t^4$ at $t = 10$ in both index origins:

```
      P
3 2 ¯5 0 1
      ⎕IO
1
      ⍳5
1 2 3 4 5
      10*¯1+⍳5
1 10 100 1000 10000
      P+.×10*¯1+⍳5
9523
```

```
      ⎕IO
0
      ⍳5
0 1 2 3 4
      10*⍳5
1 10 100 1000 10000
      P+.×10*⍳5
9523
```

In origin zero ⍳5 replaces ¯1+⍳5.

9.9 Residue: Modular Reduction

The vertical bar is used dyadically to indicate the *residue* or *remainder* of the right argument upon division by the left argument. This is a scalar function so that it extends elementwise to arrays in the usual ways:

```
      3|5                7|15               12|36
2                  1                  0

      3 7 8|10 11 12     3|10  11  12       3 7 8|10
1 4 4              1 2 0              1 3 2

      3|5.1              2.1|8              1|3.14
2.1                1.7                0.14
```

Notice that 1 | X gives the decimal part of X. The expression 3 | 5 is read the "modulo 3 residue of 5" or the "3 remainder of 5".

An outer product modulus table on the consecutive integers gives information about the structure of the integers. In particular, the columns with more than one 0 correspond to composite integers:

```
      T←2 3 4 5 6 7 8 ∘.| 2 3 4 5 6 7 8
      T
0 1 0 1 0 1 0
2 0 1 2 0 1 2
2 3 0 1 2 3 0
2 3 4 0 1 2 3
2 3 4 5 0 1 2
2 3 4 5 6 0 1
2 3 4 5 6 7 0
      0+.=T          Count the number of zeros in each column.
1 1 2 1 3 1 3
```

9.10 Example: Finite Field Tables

The simplest finite fields can be constructed by using the usual addition and multiplication of the numbers $0, 1, 2, \ldots, p-1$ modulo p, where p is prime. For example, the addition and multiplication tables of the field of order 5 can be computed by

```
      □IO
0
      5|(⍳5)∘.+⍳5
0 1 2 3 4
1 2 3 4 0
2 3 4 0 1
3 4 0 1 2
4 0 1 2 3
```

```
      5|(⍳5)∘.×⍳5
0 0 0 0 0
0 1 2 3 4
0 2 4 1 3
0 3 1 4 2
0 4 3 2 1
```

9.11 Example: Alphabetization With Mixed-Case Letters

Consider again the problem of alphabetizing a list of words, but allow both uppercase and lowercase letters. Assume □IO is 0. The alphabet here is the usual letters plus blanks, so that upon reduction modulo 27 the index of an e and an E are the same, namely 5.

```
      ALF
 abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ

      N                    ALF⍳N
Cathline          30  1 20  8 12  9 14  5
John              37 15  8 14  0  0  0  0
cat                3  1 20  0  0  0  0  0
dog                4 15  7  0  0  0  0  0
robot             18 15  2 15 20  0  0  0
jockey            10 15  3 11  5 25  0  0
Rob               45 15  2  0  0  0  0  0
```

```
      27⊥ALF⍳N                          N[↑27⊥ALF⍳N;]
 3  1 20  8 12  9 14  5        cat
10 15  8 14  0  0  0  0        Cathline
 3  1 20  0  0  0  0  0        dog
 4 15  7  0  0  0  0  0        jockey
18 15  2 15 20  0  0  0        John
10 15  3 11  5 25  0  0        Rob
18 15  2  0  0  0  0  0        robot
```

9.12 Represent

The APL primitive function *represent* or *encode*, which is designated ⊤, is used to change numbers into alternate, possibly nonuniform, base representations. The left argument is a list of the relative bases for successive digits, and the right argument is a number to be represented; in this case, the result is a vector:

```
      2 2 2 2⊤9          3 3 3⊤9          5 5 5⊤69
1 0 0 1              1 0 0            2 3 4

      2 2 2 2⊤12         3 3 3⊤15         5 5 5⊤12
1 1 0 0              1 2 0            0 2 2
```

Thus 2 2 2 2 ⊤ 9 gives a four-digit base 2 representation of 9, and 5 5 5 ⊤ 69 gives a three-digit base 5 representation of 69.

Consider a detailed evaluation of 5 5 5 ⊤ 69 from the examples given above. The residue of 69 modulo 5 is 4; which is the least significant "digit" of the result. The second digit 3 corresponds to the 5's place base 5, and the 2 corresponds to the 25's place. Thus, 25 5 1+.×2 3 4 is 69. To construct the base 5 representation for 69, use ⌊69÷25, which is 2—the most significant digit. Now 69−2×25, which is 19, needs to be represented. Next, ⌊19÷5 is 3, which is the next most significant digit, and 19−5×3 leaves the remainder 4, which is the least significant digit.

If fewer base 5 digits are requested than suffice to represent the right argument fully, a truncated representation results. If more base 5 digits are requested than are needed, the result contains leading zeros. For example,

```
      5 5⊤69             5 5 5 5⊤69          5 5 5 5 5⊤69
3 4                  0 2 3 4              0 0 2 3 4
```

The right argument need not be integral, and the base can be nonuniform so that a left argument of 4 6 5 results in a representation that has digits with weights 30 5 1. The 4 in 4 6 5 ⊤ 69 is only a place holder; any number could be used in place of the 4 without affecting the result:

```
      5 5 5⊤69.31        4 6 5⊤69         4 6 5⊤69.31
2 3 4.31             2 1 4            2 1 4.31

      8 5 5⊤69.31        8 6 5⊤69         8 6 5⊤69.31
2 3 4.31             2 1 4            2 1 4.31
```

9.13 Base Value

A complement to the represent or encode function is the *base value* or *decode* function, which is designated ⊥. It is used to compute a number that is represented by a vector. The left argument is a vector specifying the base (possibly a nonuniform base), and the right argument is the vector representing the number:

```
      5 5 5⊥2 3 4          2 2 2⊥1 1 1          3 3 3⊥1 1 1
69                    7                    13

      5⊥2 3 4              2⊥1 1 1              3⊥1 1 1
69                    7                    13

      4 6 5⊥2 1 4          4 6 5⊥1 1 1          4 6 5⊥2 1 0
69                    36                   65

      7 6 5⊥2 1 4          ¯2 6 5⊥1 1 1         0 6 5⊥2 1 0
69                    36                   65

      2⊥2 3 4              ¯2⊥2 3 4             0⊥2 3 4
18                    6                    4
```

Notice `5 5 5 ⊥ 2 3 4` gives the base 5 evaluation of `2 3 4`. For decode, a scalar left argument is uniformly extended to be a proper length vector for decode to make sense. That is unlike encode, where all the positions of the base had to be explicit.

Consider the evaluation of `4 6 5 ⊥ 2 1 4` in detail. The nonuniform base given by `4 6 5` yields weights of `30 5 1` to the digits in `2 1 4`. Therefore, yielding the same result as `30 5 1+.×2 1 4`, which is `69`. Notice also that the base does not need to be positive. For example, `¯2 ⊥ 2 3 4` is `4 ¯2 1 +.× 2 3 4`, which is `6`.

9.14 Example: Polynomial Evaluation, Finis

Consider the interpretation of $T \perp V$, where T is a scalar and V is a vector. If in common notation V has entries $p_n, p_{n-1}, \ldots, p_0$ and T is t, then $T \perp V$ is $p_n t^n + p_{n-1} t^{n-1} + \cdots + p_1 t + p_0$. This gives polynomial evaluation. Since previous representations of polynomials in this book have been with the coefficients P in increasing order, you get $T \perp \phi P$ as an expression for evaluating P at T. Once again evaluate the polynomial $P(t) = 3 + 2t - 5t^2 + t^4$:

```
      10⊥⌽3 2 ¯5 0 1        ¯2⊥⌽3 2 ¯5 0 1
9523                  ¯13

      0⊥⌽3 2 ¯5 0 1        1⊥⌽3 2 ¯5 0 1
3                     1
```

9.15 *Represent and Base Value on Arrays*

Represent and base value can be used with arrays of data as well as with multiple bases. First consider examples of represent with a vector of data 2 5 13:

```
      2 2 2 2 ⊤ 2 5 13
0 0 1
0 1 1
1 0 0
0 1 1
      5 5 ⊤ 2 5 13
0 1 2
2 0 3
```

```
      3 3 3 ⊤ 2 5 13
0 0 1
0 1 1
2 2 1
      5 5 5 5 ⊤ 2 5 13
0 0 0
0 0 0
0 1 2
2 0 3
```

Notice the columns of the results represent 2, 5, and 13 to the bases 2, 3, and 5 with the most significant digits at the top. For general arrays as right argument, the data items in the right argument are represented by the entries along the first axis of the result.

Base value decodes along the first axis of its right argument:

```
      A
0 0 1
0 1 1
2 2 1
      B
1 1 1 0
0 1 1 1
0 0 1 1
```

```
      3 3 3 ⊥ A
2 5 13
      2 2 2 ⊥ B
4 6 7 3
```

```
      3 ⊥ A
2 5 13
      2 ⊥ B
4 6 7 3
```

The situation is slightly more complicated when multiple bases are given in the left argument. For the represent function, the entries along the first axis of the left argument list the bases. Thus, the following matrix *C* refers to four-digit base 2, base 3, and base 5 representations:

```
      C
2 3 5
2 3 5
2 3 5
2 3 5
```

```
      C ⊤ 13
1 0 0
1 1 0
0 1 2
1 1 3
```

```
      C ⊤ 15
1 0 0
1 1 0
1 2 3
1 0 0
```

For the base value function, the entries along the last axis give the bases. If *D* and *E* are matrices, *D* ⊥ *E* yields a matrix whose entries come from decoding the columns of *E* with bases taken from rows of *D* in all possible combinations. This is very similar to the row-by-column principle of inner

products. Here let $D \leftarrow \boxtimes C$ and $E \leftarrow C \top 15$:

```
      D                E                D⊥E
2 2 2 2          1 0 0          15  8  6
3 3 3 3          1 1 0          40 15  9
5 5 5 5          1 2 3         156 35 15
                 1 0 0
```

For example, the 35 in $D \perp E$ is the result of `5 5 5 5 ⊥ 0 1 2 0`, which is the base 5 evaluation of `0 1 2 0`. The diagonal entries are all 15 since they are the result of encoding and decoding 15 with the same base.

As a last example consider the problem of converting between seconds and hours, minutes, and seconds. Below several conversions are given. Two of them are 60 seconds is 0 hours, 1 minute, and 0 seconds and 4000 seconds is 1 hour, 6 minutes, and 40 seconds:

```
      A←24 60 60 ⊤ 60 345 4000 6300
      A
0  0  1  1
1  5  6 45
0 45 40  0
```

Notice that the most significant unit (hours) appears at the top of the column.

Next base value can be used to convert back to seconds:

```
      24 60 60 ⊥ A
60 345 4000 6300
```

9.16 Atomic Vector

APL provides the vector $\square AV$, which is a list of all the characters available in the APL system. Many of these characters are not displayable; you should experiment with $\square AV$. In this section $\square IO$ is assumed to be 0.

The first example computes the indices of some APL characters in $\square AV$; the second shows indexing of $\square AV$ to get characters:

```
      ⎕AV⍳'↑AB*Δ'                ⎕AV[30 30 71 86]
71 86 87 30 112          **↑A
```

Some of the nondisplayable characters are used for invoking control features. For example, $\square AV[156]$ designates a carriage return. Thus, if $SKIP \leftarrow 5\rho\square AV[156]$, the vector $SKIP$ may be manipulated as a character vector of length 5, but in response to executing $SKIP$ the APL system displays five carriage returns—that is, it skips 5 lines.

Use of $\square AV$ as a reference vector of characters is illustrated in the next section.

9.17 Example: Encoding a String Into Binary

Consider the problem of representing a string of text in binary. Let $\square IO \leftarrow 0$ and

$V \leftarrow$ '$\alpha * \uparrow 035ABC * \omega$'.

First, $\square AV\iota V$ is used to represent V as a numeric vector of indices in the range 0 to 255. Then those integers are represented using an eight digit binary number:

```
        V
α*↑035ABC*ω
        □AV⍳V
42 30 55 140 143 145 86 87 88 30 46
        A←(8⍴2)⊤□AV⍳V
        A
0 0 0 1 1 1 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 0
1 0 1 0 0 0 0 0 0 0 1
0 1 1 0 0 1 1 1 1 1 0
1 1 0 1 1 0 0 0 1 1 1
0 1 1 1 1 0 1 1 0 1 1
1 1 1 0 1 0 1 1 0 1 1
0 0 1 0 1 1 0 1 0 0 0

        2⊥A
42 30 55 140 143 145 86 87 88 30 46
        □AV[2⊥A]
α*↑035ABC*ω
```

Notice that it was easy to reinterpret the binary "message" A as the original text string.

9.18 Example: Hamming Code for Error Correction

Error-correcting codes are used to partially repeat the information in a message so that errors in transmission can be detected and corrected. The goal is to repeat the information in such a way that only small amounts of additional information need to be transmitted. Here the implementation of the simplest Hamming Code is considered. It can identify and correct a single error in a four-bit message by sending three additional bit checks. See texts such as Tucker (1988) and Pleuss (1982) for motivation of this algorithm.

The message 1 1 1 0 is encoded using matrix multiplication by Q (given below) and reduction modulo 2:

$$\underbrace{\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Q} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 1 \\ 2 \\ 1 \\ 1 \\ 0 \end{bmatrix} \equiv \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \text{ (modulo 2)}$$

With *Q* as above and *MES*←1 1 1 0, a direct APL computation of the resulting encoded message is given by *EMES*←2 | *Q*+ . ×*MES*:

```
      EMES
0 0 1 0 1 1 0
```

Actually, it is better to replace the arithmetic of + . × followed by reduction modulo 2 by logical functions. In particular, for binary numbers multiplication modulo 2 is the same as "logical and", and addition modulo 2 is the same as "logical not equals". Those facts follow from the tables:

α×ω	0	1
0	0	0
1	0	1

2 | α×ω	0	1
0	0	0
1	0	1

α∧ω	0	1
0	0	0
1	0	1

α+ω	0	1
0	0	1
1	1	2

2 | α+ω	0	1
0	0	1
1	1	0

α≠ω	0	1
0	0	1
1	1	0

Therefore, matrix multiplication modulo 2 of binary arrays can be accomplished using the ≠ . ∧ inner product.

In some implementations of APL, the use of ≠ . ∧ provides a dramatic increase in speed. Using this inner product,

```
      EMES←Q≠ . ∧MES
      EMES
0 0 1 0 1 1 0
```

Notice that with □*IO*←0, rows 2, 4, 5, and 6 of *Q* are rows of the identity matrix. So it is easy to recover the original message by looking at the entries 2, 4, 5, and 6:

```
      EMES[2 4 5 6]
1 1 1 0
```

Now consider the possibility of corruption of the message during transmission. Suppose the third bit is incorrectly sent so that the received message *RMES* is given by

```
      RMES
0 0 0 0 1 1 0
```

The error can be recognized by multiplying modulo 2 by the matrix *M*, given below; again ≠ . ∧ accomplishes the matrix multiplication and reduction:

```
      M
1 0 1 0 1 0 1
0 1 1 0 0 1 1
0 0 0 1 1 1 1
```

```
      M≠ . ∧EMES
0 0 0
      M≠ . ∧RMES
1 1 0
```

The fact that *M*≠ . ∧*EMES* is the zero vector indicates that *EMES* does not appear to contain an error. Since *M*≠ . ∧*RMES* is not the zero vector, some of the bit checks have failed, so there is an error. *M* is constructed so that the error (assuming there is just one) is in the position of *RMES*, which corresponds to the column of *M* that equals *M*≠ . ∧*RMES*, column 2 in this example. (Recall □*IO*←0.)

Here *M*≠ . ∧*RMES* and the columns of *M* are interpreted as base 2 numbers and the column index of *M*≠ . ∧*RMES* in *M* is found by use of dyadic iota. This trick for finding the index of the desired

column depends on the fact that $2 \perp M$ has no repeated entries:

```
      (2⊥M)⍳2⊥M≠.∧RMES      Find index of error.
2
      RMES[2]←~RMES[2]       Correct the error.
      RMES                   Corrected received message.
0 0 1 0 1 1 0
      RMES[2 4 5 6]          Original four-bit message.
1 1 1 0
```

9.19 Example: Hamming Code on Natural Text

Consider the problem of sending the message *NMES*←'*WE HAVE FOUND* $3,000,000!' via the Hamming Code. Since the Hamming Code used here acts on four-bit messages and □*AV* has 256 characters so that 8 bits are used to represent a character in □*AV*, it is convenient to represent the 25-character message *NMES* as a three-dimensional 4-by-2-by-25 Boolean array. The Hamming code will act along the first axis.

Each character is represented as an integer in the range 0–255; then that integer is represented as two base 16 digits. Each of those digits is then represented as a four-bit Boolean word along the first axis. In the example below the character '*N*' in '*FOUND*' is underlined and all the data resulting from that letter are underlined throughout; the computations on '*N*' are also summarized in a flow chart (Figure 9.1):

```
      NMES
WE HAVE FOUND $3,000,000!
      □AV⍳NMES
108 90 152 93 86 107 90 152 91 100 106 99 89 152 4 143 47 140
      140 140 47 140 140 140 48
      16 16⊤□AV⍳NMES
 6  5  9  5  5  6  5  9  5  6  6  6  5  9  0  8  2  8  8  8  2
       8  8  8  3
12 10  8 13  6 11 10  8 11  4 10  3  9  8  4 15 15 12 12 12 15
      12 12 12  0
      MES←2 2 2 2⊤16 16⊤□AV⍳NMES
      MES
0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 0 1 1 1 0 1 1 1 0
1 1 1 1 0 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 1 1 1 1 0

1 1 0 1 1 1 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0

1 0 0 0 0 1 0 0 0 1 1 1 0 0 0 0 1 0 0 0 1 0 0 0 1
0 1 0 0 1 1 1 0 1 0 1 1 0 0 0 1 1 0 0 0 1 0 0 0 0

0 1 1 1 1 0 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1
0 0 0 1 0 1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 1 0 0 0 0
```

The natural message *NMES* has been coded into the binary message *MES*. Now *MES* is encoded with the Hamming encoding matrix *Q*. The under-dots indicate where the message will be corrupted during transmission:

```
      EMES←Q≠.∧MES
      EMES
1 0 0 0 0 1 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 1 1
0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 1 1 0 0 0 1 0 0 0 0

1 1 0 1 1 1 1 0 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 0
1 0 1 0 1 1 0 1 1 0 0 0 0 1 0 1 1 1 1 1 1 1 1 1 0

0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 1 1 1 0 1 1 1 0
1 1 1 1 0 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 1 1 1 1 0

0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0
1 1 0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0

1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0

1 0 0 0 0 1 0 0 0 1 1 1 0 0 0 0 1 0 0 0 1 0 0 0 1
0 1 0 0 1 1 1 0 1 0 1 1 0 0 0 1 1 0 0 0 1 0 0 0 0

0 1 1 1 1 0 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1
0 0 0 1 0 1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 1 0 0 0 0
```

Now consider the possibility that the message is corrupted during transmission and hence incorrectly received. This is simulated below by letting *RMES* be the encoded message but forcing 12 entries to be 1s; those entries are indicated with under dots in *EMES*. Notice that reinterpreting *RMES* as natural language text does not restore the original message — the message was corrupted. Although several errors are introduced into *RMES*, these can all be corrected since they are in separate seven-bit messages given along the first axis:

```
      RMES←EMES

      RMES[2;1;3+⍳12]←1                    Introduce errors.

      ⎕AV[16⊥2⊥RMES[2 4 5 6;;]]            Decode RMES; observe errors.

WE HIVE FWUVD ■3,000,000!
```

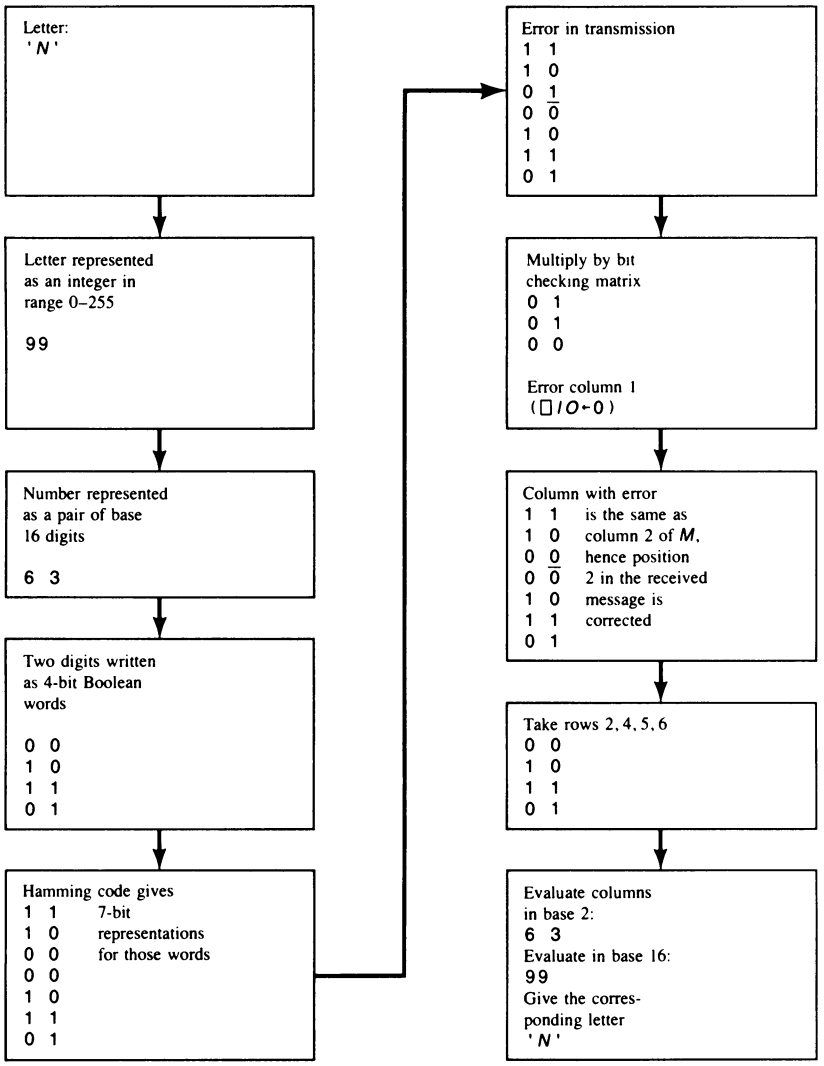


Figure 9.1 Summary of coding and decoding 'N'.

Next, the received message is multiplied by the bit check matrix *M* as in Section 9.18:

```
      M
1 0 1 0 1 0 1
0 1 1 0 0 1 1
0 0 0 1 1 1 1
      M≠.∧RMES
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The presence of 1s shows that many of the bit checks failed. These are in the columns where a 0 was set to 1 and where the errors in the decoded *RMES* were observed. As in Section 9.18, the column indices in *M* of those errors are computed. The same trick of evaluating the columns of *M* and *M≠.∧RMES* in base 2 allows dyadic iota to give the column indices of *M* where those columns of *M* equal the corresponding columns of *M≠.∧RMES*. An index of 7 corresponds to no error. The array *COR* contains a 1 in the positions in which *RMES* need to be corrected. The corrections are done by adding *COR* to *RMES* modulo 2, which is implemented via the not equals function:

```
      (2⊥M)⍳2⊥M≠.∧RMES
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
7 7 7 7 2 7 7 7 7 2 7 2 7 7 2 7 7 7 7 7 7 7 7 7 7

      COR←(⍳7)∘.=(2⊥M)⍳2⊥M≠.∧RMES
      COR
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
      CMES←RMES≠COR
      ⎕AV[16⊥2⊥CMES[2 4 5 6;;]]
WE HAVE FOUND $3,000,000!
```

After the correction is made, the original natural text message can be successfully recovered from the received message.

Exercises

1. Consider D←12 19 26 13 9 14.
   (a) Give an APL expression to order D in increasing order.
   (b) Give an APL expression to order D in decreasing order.
2. In Section 9.1 the application of grade up to a vector was considered. The expression ⍋V results in a list of indices beginning with the index of the smallest element, and so on. The expression ⍋⍋V gives the rank in increasing order of the elements of V. Describe in words the meaning of the following expressions (suggestion: experiment):
   (a) ⍋⍋⍋V (b) ⍋⍋⍋⍋V (c) ⍋⍒V
3. For a general array A, what is the shape of ⍋A?
4. Consider the examples of reordering and ranking the data on grade point averages given in Section 9.3. Interpret verbally the examples given in that section.
5. Consider the following table of computer resource requirements of three algorithms 1, 2, 3 on two problems 1, 2:

Algorithm	Prob	Time (sec)	Space (MB)
1	1	12	0.6
2	1	24	0.5
3	1	15	2.0
1	2	22	1.5
2	2	36	1.6
3	2	15	1.5

   Assuming that the matrix M contains the information given in the table, give APL expressions to reorder that information in the following ways.
   (a) Fastest runs appear first, with ties decided by looking at space.
   (b) Keep the data ordered by problem but reorder by time within the problems.
   (c) Order by algorithm number with the least-space solution first.
6. For the same data as in Exercise 5, give an APL expression that gives the data of the problem and attaches a column that ranks the rows according to fastest runs with ties decided by a least-space requirement.
7. Write an APL function that takes a list of words as the rows of a matrix (uppercase only) as the argument and results in the list alphabetized.
8. Write an APL function that takes a list of words as the rows of a matrix involving both uppercase and lowercase letters and that alphabetizes so that the capital letter always comes before the lowercase version of the same letter. For example, allen is after Allen.
9. Let V←2 9 2 8 3 7 9 5 and assume ⎕IO←0. What are
   (a) V⍳2 3 4 5 (b) V⍳⍳9 (c) V⍳V
10. Let ALF←' ABCDEFGHIJKLMNOPQRSTUVWXYZ' and ⎕IO←0. What are
    (a) ALF⍳'HELLO' (b) ALF⍳'ALF' (c) ALF⍳ALF

11. Consider $A \iota B$ for arrays A and B.
    (a) What are the requirements on A and B so $A \iota B$ makes sense?
    (b) In that case, what is the shape of $A \iota B$?
12. Describe verbally the result of $V[\Downarrow V] \iota V$. Experiment with
    $V \leftarrow 12\ 3\ 4\ 8\ 4\ 4\ 11\ 12\ 9\ 8$.
13. Write an APL function *NUB* that takes a vector argument and results in the nub of the vector.
14. Write an APL function *FFREQ* (fast frequency) that takes a vector argument and results in a two-row matrix giving the ordered nub of the argument in the first row and the corresponding frequencies in the second row.
15. Write a function *POLYVAL* that evaluates a polynomial at several points. Use $\square IO \leftarrow 0$. (Hint: See Section 9.8.)
16. Make slight modifications to the *HIST* in Section 7.8 so that it works with $\square IO \leftarrow 0$.
17. Let T be a scalar t and P and Q be vectors representing polynomials $P(t)$ and $Q(t)$ such that coefficients are in increasing order. Write an APL expression that evaluates the composite of the polynomials at t: $P(Q(t))$.
18. An APL expression is $\square IO$ independent if it performs in the same manner regardless of the value of $\square IO$. Write an APL function for polynomial evaluation that is $\square IO$ independent. (Hint: The function may refer explicitly to $\square IO$ but it should give the correct answer regardless of whether $\square IO$ is 0 or 1.)
19. Give an APL expression and its result for the number of divisors of the integers 2–100; for example, 8 has 4 divisors (1, 2, 4, and 8).
20. Give an APL expression that gives the primes less than or equal to 100. (Hint: See Exercise 19.)
21. (a) Give APL expressions to compute addition and multiplication tables modulo 6. Also give the tables.
    (b) Do the same for modulo 7.
    (c) Describe verbally how these tables are different.
22. Write an APL function *SAWTOOTH* that extends the absolute value function on $[-\pi, \pi]$ periodically to the real line. Recall that π is computed by $\circ 1$.
23. Write an APL function that alphabetizes the rows of its matrix argument without regard for the case of the letters. Let the digits 0, 1, 2, ..., 9 be legal characters that follow the letters in the alphabet.
24. (a) Give an APL expression that yields a matrix having as columns all the four-digit binary numbers.
    (b) Give an APL expression that yields a matrix having as columns all the three-digit base 3 numbers.
25. Write an APL function *NUMBASE* that has arguments K and N and that yields all of the K-digit base N numbers in the columns of its result.
26. Give an expression for and compute the number of four-digit base N palindromic numbers for N equal to 2 through 5. (Hint: Use *NUMBASE* from Exercise 25.)
27. Consider $V \top B$ for a vector V.
    (a) What are conditions on ρV and ρB so that $V \top B$ makes sense?
    (b) What is the shape of $V \top B$?
28. Consider $V \perp B$ for a vector V.
    (a) What are conditions on ρV and ρB so that $V \perp B$ makes sense?
    (b) What is the shape of $V \perp B$?
29. (a) Write a function *S2HMS* that takes a list of times in seconds and results in a matrix giving the corresponding hours, minutes, and seconds in the columns of the result. Test your function on 50 500 5000 50000.
    (b) Write a function *HMS2S* that takes a matrix with hours, minutes, and seconds given in the columns and results in a vector giving the corresponding seconds. Test your function on the result of (a).

(c) Write a function *H2HMS* that takes a list of times in hours (fractional allowed) and results in a matrix giving the corresponding hours, minutes, and seconds in the columns of the result. Test your function on `0.25 1 1.5 2.75 3.1415`.

(d) Write a function *HMS2H* that takes a matrix with hours, minutes, and seconds given in the columns and results in a vector giving the corresponding hours. Test your function on the result of (c).

30. Write an APL function *HAM* that takes natural text as input and results in a binary array encoded with the Hamming Code considered in Section 9.19.

31. Write an APL function *CORHAM* that takes an array encoded with the Hamming code and that may contain errors and that corrects the errors.

32. Write an APL function *UNHAM* that takes a binary array encoded with the Hamming code and results in the natural text it represents.

33. Use the function *HAM* to encode the message `'THE SEMESTER IS TOO LONG'` as the variable *EMES*. Modify the received message in several ways. Let

```
RMES1←RMES2←RMES3←RMES4←RMES5←RMES6←EMES
RMES1[4;0;⍳10]←1 ◊ RMES2[5;0;⍳10]←1 ◊ RMES3[4 5;0;⍳10]←1
RMES4[3;0;⍳10]←1 ◊ RMES5[5;0 1;⍳10]←1 ◊ RMES6[1 3;0;⍳10]←1
```

Use *UNHAM* to decode the six messages before and after using *CORHAM*. Explain why some errors did not need to be corrected and why some could not be corrected.

10

More Function Definition

This chapter provides tools useful for writing complex APL functions and "programs". They include forms of branching, line labels, recursive functions, methods for input and output, format, and the powerful function execute. The chapter also gives many examples.

The conventional notion of a computer program differs from the kinds of APL functions discussed previously. Computer programs lack the essential mathematical nature that those APL functions had; namely, those functions had explicit arguments and produced values or results that are automatically passed along as arguments for the next function. User-defined functions of "programming type" are introduced in this chapter.

10.1 Functions for Programming Uses

All user-defined functions previously considered produced a result or value that was passed on as an argument for the next function to the left or was displayed in the session log. Additionally, all those functions had arguments. Those features, which are also shared by the APL primitive functions, make them extremely useful but are not needed in some "programming" applications.

The desired effect of a function might be, for example, the updating of the value(s) of one or more global variables (variables not localized in a function). A function might be constructed to invoke several other functions or effect an organized display of data. APL provides function types suited for such programming needs. These functions may or may not have an explicit argument and may or may not automatically pass on a result.

The form of the function header determines whether a defined function does or does not automatically pass a value to the next function and whether the function does or does not require explicitly specified arguments. If the variable assignment, $Z\leftarrow$, is omitted from the header, the function does not pass on a result that could be used as an argument for another function. Similarly, if no argument variables are placed on either side of the function name in the header, the function does not require explic-

itly specified arguments. Simply omitting one or both of those parts of the header causes the defined function to lack the corresponding feature. A function that automatically passes a value to the next function is said to "return a result". Table 10.1 gives examples of header styles for a function *F*.

Table 10.1 Examples of header styles for a function *F*.

	Number of Arguments		
	Dyadic 2	Monadic 1	Niladic 0
Returns a result	*Z←A F W*	*Z←F W*	*Z←F*
No result passed	*A F W*	*F W*	*F*

Comments can be helpful reminders in lines of APL. Anything following the *lamp* symbol ⍝ on a line is a comment and is not executed by APL. Some implementations permit comments only on separate lines.

The function *COUNT* is a simple example of a function that neither takes an argument nor returns a result. Execution of this function requires that *C* be a global numeric variable in the workspace. Notice the absence of output variable assignment and arguments in the header:

```
     ∇ COUNT
[1]    C←C+1  ⍝ EACH COMPONENT OF C IS INCREMENTED BY 1.
     ∇

       C       ⍝ SHOW CURRENT VALUE OF C
17
       COUNT   ⍝ EXECUTE THE FUNCTION; NO VALUE IS DISPLAYED
       C       ⍝ OBSERVE THE NEW VALUE OF C
18
```

In contrast to all previous functions, which were either monadic or dyadic, the function *COUNT*, which has no explicit argument, is termed *niladic*.

If a value is computed or a character array is generated as a result of the last function execution of a line of APL, that result is displayed in the session log. This feature has been used often in this book in immediate execution mode, but it applies equally in executing a line of a defined function. Here is an example in a niladic function:

```
     ∇ HI
[1]    'HELLO WORLD'
     ∇

       HI
HELLO WORLD
```

Although the function *HI* caused output, it does not return a result. For instance, ⍴*HI* would produce an error.

10.2 Example: Descriptive Statistics

The next example, *STATISTICS*, has a vector argument but does not return a result; that is, it does not pass on a value usable as an argument for another function. This function invokes several functions from Chapter 7; definitions of these functions are given there, except for *MODE*, which is an answer to Exercise 7.12. Again, observe the header:

```
     ∇ STATISTICS V
[1]    'RANGE: ' ◊ RANGE V
[2]    'MODE(S): ' ◊ MODE ⌊.5+V
[3]    'MEDIAN: ' ◊ MEDIAN V
[4]    'AVERAGE: ' ◊ AVG V
[5]    'STANDARD DEVIATION: ' ◊ SD V
[6]    'HISTOGRAM: ' ◊ HIST ⌊.5+V
     ∇
```

Notice that the data in *V* are rounded to integers before applying the functions *MODE* and *HIST*, since these functions were written for integral data only. Here is a sample execution of *STATISTICS*:

```
      V←5.5 4 3.1 3.8 7 6.1 5.8 10 10.9 7.1 6 10 9.2 12 7 11 9.9
      STATISTICS V
RANGE:
3.1 12
MODE(S):
6
MEDIAN:
7
AVERAGE:
7.552941176
STANDARD DEVIATION:
2.76385547
HISTOGRAM:
    *
    **   *
  *  **   **
**  **  ****
```

When the quantity of output is large, care in formatting makes the data more attractive in appearance and more readily comprehensible. Formatting is discussed in Section 10.15.

10.3 Branching

One of the powerful devices in user-defined functions is *branching* — the capacity for directing the subsequent flow of function execution depending on conditions at the branch point. In Section 3.9, conditional branching was introduced and used to execute simple loops. The statement $\rightarrow 4\times K<N$ was used in that context; it causes execution to branch to line 4 if the counter K is less than N; otherwise

the branch is "to zero"; that is, execution terminates. In Chapter 6, branching was used outside the context of loops with counters. For example, `→2+2×1=M[B;C]` from the function *SIM* in Section 6.3 causes a branch to line 4 if the matrix element `M[B;C]` is 1; otherwise the branch is to line 2. In these examples a logical function is involved in calculating one of two possible line numbers or the number 0.

Besides branching to a line number, which may be given as a scalar or a one-element vector, APL recognizes branching to the empty vector `⍳0`. The interpretation of `→⍳0` is not to branch at all in the usual sense but rather to proceed with execution of the next succeeding statement of the function. Thus, if *V* can take as values either a line number or `⍳0`, the expression `→V` in a function effects a branch to either the specified line or else the next line of the function.

One way to generate either a number or `⍳0` is by compression (Sections 7.9 and 8.5):

```
      1/3                0/3                1/0
3                  (empty vector)     0

      ⍴1/3               ⍴0/3               ⍴1/0
1                  0                  1
```

Note, `0/3` yields the empty vector, giving no visible display.

Thus, the expression `→(N<10)/3` causes a branch to line 3 if *N* is less than 10, but causes execution of the next succeeding line of the function if *N* is not less than 10. The expression `→(N=M)/0` causes termination of function execution if *N* equals *M* but produces execution of the next function line if *N* does not equal *M*.

Another way of causing a conditional branch is by using the residue function. The residue function (Section 9.9) is used to find the remainder of a division. This is indicated with dyadic use of a vertical bar. The result is the remainder upon dividing the right argument by the left argument:

```
      2|5                2|8                2|¯7
1                  0                  1

      3|5                4|5                1|7
2                  1                  0
```

Hence, the expression `→(2|N)/4` causes a branch to line 4 if *N* is an odd integer but causes execution of the next line if *N* is an even integer.

Consider the function *REMOVE2S*. This function is designed to take a positive integer *N* as argument and return the largest factor of *N* that has no factor of 2 in it; for example, `REMOVE2S 60` should yield 15:

```
     ∇ Z←REMOVE2S N
[1]    →(2|N)/4          ⍝ BRANCH TO LINE 4 IF N IS ODD
[2]    N←N÷2
[3]    →1
[4]    Z←N
     ∇
```

In line 1, if the 2 remainder of *N* is 1 (i.e., *N* is odd), then the compression yields 4 and execution skips to line 4, after which execution is complete and the value of the function is passed on to the next function or is displayed in the session log. But if 2 exactly divides *N* leaving remainder 0, then

the compression yields ⍳0 and there is no branch; execution proceeds to the next line, line 2. *N* has one factor of 2 removed from it in line 2; then the unconditional branch →1 returns execution to line 1. The new value of *N* is tested for oddness and the cycle continues until *N* is odd; then the branch to line 4 leads to the completion of function execution:

```
      REMOVE2S 60          REMOVE2S 1000          REMOVE2S 1728
15                    125                    27
```

10.4 Line Labels

A *line label* provides a name for a function line. Labeling a line is effected by beginning the line with the label followed immediately by a colon. ***A line label has a numerical value*** within the function; namely, the number of the line it labels.

The names *TEST* and *END* will be added as line labels to the function *REMOVE2S* of Section 10.3. Their values in this first example are 1 and 4, respectively, since lines 1 and 4 are labeled by them:

```
    ∇ Z←REMOVE2S N
[1]   TEST:→(2|N)/END     ⍝ BRANCH TO "END" IF N IS ODD
[2]    N←N÷2
[3]    →TEST
[4]   END:Z←N
    ∇
```

Notice the syntax of label usage. This version of the function with line labels is identical in execution to the original version. Line labels are automatically localized to the function containing them. An advantage of the use of line labels is that modifications of a function that affect the numbering of lines in the function will not disturb the flow of execution determined by branches to labeled lines. Thus, there is no need to renumber branching "targets" to match the new line numbers.

Here is a specific example: The function *REMOVE2S*, as presently written, would go into an "infinite loop" (i.e., would cycle endlessly through lines 1, 2, 3, 1, 2, 3, . . .) in response to the argument *N* being 0. To avoid getting into this infinite loop, you could insert →(Z←N=0)/0 as a new first line of the function. If *N* equals 0, this new line would assign *Z* the value 1 and terminate execution; but if *N* is not 0 the compression would yield ⍳0 and function execution would proceed to the next line. After inserting the new line the revised function is

```
    ∇ Z←REMOVE2S N
[1]   →(Z←N=0)/0          ⍝ TREAT CASE N=0
[2]   TEST:→(2|N)/END     ⍝ BRANCH TO "END" IF N IS ODD
[3]    N←N÷2
[4]    →TEST
[5]   END:Z←N
    ∇
```

The values of *TEST* and *END* are now 2 and 5, but the branching to those named lines is unchanged; ***no*** adjustment of line numbers is necessary. For positive *N* the function behaves as before:

```
      REMOVE2S 0           REMOVE2S 1728          REMOVE2S 512
1                     27                     1
```

Good programming style benefits by use of line labels. Modification of a function that affects the numbering of lines will not disturb the flow of execution determined by the branches to labeled lines. Line labels help make functions with branches more easily interpretable. A label signals the reader to expect a branch to the labeled line. A suggestive name is helpful for understanding why the branch is used.

10.5 Examples: Aggregating Preferential Ballots

Almost all organizations and political subdivisions elect officers or representatives or choose among alternate courses of action. A few of the simplest methods for arriving at a group decision based on the preferences of individual members will be described and interpreted as APL functions. The examples have several alternatives, one of which is to be selected. Each voter expresses his or her preference among the alternatives by ranking them first, second, and so on.

In the example in Figure 10.1, the voter has given the highest ranking to alternative 3, next highest to alternative 4, and lowest to alternative 1.

The individual voters' ballots are combined to form a matrix named *B* called the Ballot; the capital B in Ballot is used to distinguish it from the individual ballots. Each column of *B* is the preference ranking of one of the voters. Row *J* of *B* contains the rankings given to alternative *J* by the voters. Thus, *B*[*J*;*K*] is the ranking that voter *K* assigned to alternative *J*. Two sample Ballots, *B* and *C*, recording the preferences of 11 voters on 4 alternatives are

```
      B                                   C
4 2 1 4 2 4 3 2 1 4 3     3 2 4 3 2 4 2 2 4 3 3
3 4 3 1 3 2 1 3 2 3 1     2 3 3 4 4 2 4 3 2 4 4
1 1 4 2 1 3 2 4 3 1 2     1 4 1 2 3 1 3 4 1 2 1
2 3 2 3 4 1 4 1 4 2 4     4 1 2 1 1 3 1 1 3 1 2
```

Notice that the individual ballot illustrated in Figure 10.1 comprises the first column of Ballot *B*.

Majority. The majority choice is the alternative that receives more than half the total number of first-place rankings. Of course, the Ballot might not determine a majority choice. The function *MAJORITY* has *B* as an argument and returns the number of the majority choice alternative; if there is none, it returns 0:

```
     ∇ Z←MAJORITY B;N;S;T
[1]    Z←0 ◊ N←1↓⍴B
[2]    →((N÷2)≥T←⌈/S←+/1=B)/0
[3]    Z←S⍳T
     ∇
```

Alt. 1	4
Alt. 2	3
Alt. 3	1
Alt. 4	2

Figure 10.1 An individual's ballot.

N is the number of voters. In line 2, the number of first-place rankings for each alternative is counted by the expression `+/1=B`; those numbers are stored as the vector *S*. If the largest component of *S*, which is named *T*, is not more than half the number of voters, the branch is to 0 (execution ends) and the function returns 0. Otherwise, the branch is to `⍳0`, and in line 3 the *S* index of *T*, which is the index of the majority choice, is returned by the function. Applying *MAJORITY* to the Ballots *B* and *C* shown above yields

```
      MAJORITY B                    MAJORITY C
0                                   4
```

Ballot *B* does not determine a majority choice.

Plurality. The plurality choice is the alternative that receives the most first-place rankings. If two or more alternatives tie with the greatest number of first-place rankings, no plurality choice is determined and the function *PLURALITY* returns 0:

```
    ∇ Z←PLURALITY B;S;T
[1]   Z←0
[2]   →(1<+/S=T←⌈/S←+/1=B)/0
[3]   Z←S⍳T
    ∇
```

In line 2, the first-place rankings for each alternative are counted and stored in *S*. The maximum component of *S* is assigned to *T*, and the number of components in *S* that equal *T* is determined. If there is more than one such component, the branch is to 0 and the function returns 0. But if there is only one component of *S* that equals *T*, the branch is to `⍳0`, and in line 3 the *S* index of that one component, which is the plurality choice, is the number returned by the function. Applying *PLURALITY* to Ballots *B* and *C* yields

```
      PLURALITY B                   PLURALITY C
3                                   4
```

Plurality with runoff. Runoffs between leading alternatives are commonly used when there is no majority choice, but runoff procedures vary. In the present implementation, the two alternatives receiving the greatest number of first-place rankings become the only alternatives in a runoff ballot count. The relative rankings of the runoff alternatives on each individual ballot determine the alternative for which that ballot counts in the runoff. For example, if one of the runoff alternatives received a 2 ranking on a particular ballot and the other a 4 ranking, that ballot would be counted for the alternative that got the 2 ranking. If there is not a unique pair of alternatives with the greatest number of first-place rankings, this implementation does not make a choice and the function *RUNOFF* returns 0.

```
    ∇ Z←RUNOFF B;N;P;R;S
[1]   Z←0 ⋄ N←1↓⍴B ⋄ R←⍒S←+/1=B
[2]   →(S[R[2]]=S[R[3]])/0          ⍝ TEST FOR UNIQUE LEADING PAIR
[3]   P←B[R[1];]+.<B[R[2];]         ⍝ P IS NO. BALLOTS AWARDED R[1]
[4]   →(P>N÷2)/END                  ⍝ TEST WHETHER R[1] WINS
[5]   Z←R[2]×(N÷2)<N-P              ⍝ TEST WHETHER R[2] WINS
[6]   →0
[7]   END:Z←R[1]
    ∇
```

N is the number of voters. In line 1, the first-place rankings for each alternative are counted and stored in *S* and the downgrade of *S* is stored in *R*. *R* has the alternative numbers (indices) listed in decreasing order of the number of first-place rankings received. If the second and third largest numbers of first-place rankings are equal so that there is no unique pair of leading alternatives, then the branch in line 2 is to 0; no choice is made and *RUNOFF* returns 0. Otherwise, in line 3 the relative rankings of the two leading alternatives are compared and the counts of ballots awarded to those alternatives are *P* and *N*−*P*. If *P* is greater than *N*÷2, the branch in line 4 is to *END*, where the number of the alternative receiving *P* ballots in the runoff, *R*[1], becomes the value returned by the function. If *P* is not greater than *N*÷2, line 5 is executed. If *N*−*P* is greater than *N*÷2, the corresponding alternative number, *R*[2], is returned by the function. Note that if *P* equals *N*÷2, no runoff choice is determined and the function returns 0.

Applying *RUNOFF* to Ballot *B* yields

```
      RUNOFF B
2
```

10.6 Multioption Branches

Branches can have more than two possible branch "targets". Thus, if *N* is a positive integer, the expression →4+3|*N* would cause a branch to either line 4, line 5, or line 6, depending on the 3 remainder of *N*. If *L*/*NEA*, *L*/*NEB*, and *L*/*NEC* are line labels within a function, they have numerical values that can be catenated. Then the expression →(*W*=⍳3)/*L*/*NEA*,*L*/*NEB*,*L*/*NEC* would cause a branch to *L*/*NEA* if *W* equals 1, *L*/*NEB* if *W* equals 2, and *L*/*NEC* if *W* equals 3; otherwise, the branch is to ⍳0. Similarly, the expression

```
→(¯1 0 1=×W)/L/NEA,L/NEB,L/NEC
```

would cause a branch to *L*/*NEA* if *W* were negative, *L*/*NEB* if *W* were 0, and *L*/*NEC* if *W* were positive. This last example, that uses signum, arises quite naturally in the root approximating function defined next.

10.7 Example: The Bisection Method

The bisection method is a simple method for approximating a root of $f(x) = 0$ on an interval $[a, b]$ if it is known that f is continuous there and that $f(a)$ and $f(b)$ have opposite signs. Although usually slower than Newton's method, the bisection method has the advantage of being surefire. The idea is to look at the sign of f at the midpoint z of $[a, b]$. If $f(z) = 0$, then z is a root; otherwise, choose one of the intervals $[a, z]$ or $[z, b]$ on which f changes sign (and thus has a root) to be the new interval on which the process is to be repeated. Repeated bisection leads to better and better root approximations.

Let *F* be a user-defined function, continuous on an interval containing *A* and *B* with *A* less than *B*. Suppose *F A* and *F B* have opposite signs. Let *TOL* be a global variable that prescribes the required tolerance of the root approximation. Then the dyadic function *B*/*SECTMETHOD* applies the bisection method and yields a root approximation to within the tolerance specified by *TOL*. Note, particularly, the three-way branch in line 3 and the assignment to quad, □←*Z*, in line 1, which will cause

each successive approximation to be displayed:

```
     ∇ Z←A BISECTMETHOD B
[1]   BIS: ⎕←Z←.5×A+B      ⍝ MIDPOINT GIVES APPROXIMATION
[2]    →(TOL>.5×B-A)/0     ⍝ IS APPROX. WITHIN TOLERANCE
[3]    →(¯1 0 1=×(F A)×F Z)/BEND,0,AEND
[4]   BEND:B←Z ◊ →BIS
[5]   AEND:A←Z ◊ →BIS
     ∇
```

Illustration. Approximate the zero of the continuous function $f(x) = x - e^{-x}$ to within .005 of a unit. Note that $f(0) = {}^{-}1 < 0$ and $f(1) = 1 - e^{-1} > 0$. First, define *F* and *TOL*:

```
     ∇ Z←F X
[1]    Z←X-*-X
     ∇

       TOL←.005
       0 BISECTMETHOD 1
0.5
0.75
0.625
0.5625
0.59375
0.578125
0.5703125
0.56640625
0.56640625
```

Thus, the solution of $x - e^{-x} = 0$ is within .005 of 0.56640625.

10.8 Recursive Functions

User-defined functions frequently invoke other user-defined functions. An APL user-defined function can call upon itself. As with all function calls, a self-call is effected simply by using the function name. A function that in its definition calls upon itself is termed a recursive function.

The first example of a recursive function given here is for illustrative purposes only; it mimics simple multiplication via repeated addition using the fact that *N*×*A* can be computed by *A*+(*N*−1)×*A*. The dyadic function *TIMES* gives the same result as *N*×*A*, where *N* is a positive integer and *A* is any numeric array. Note that in line 3 the function invokes itself but with the left argument reduced by 1. It is instructive to think through the execution of this function carefully for the *N* values 1, 2, and 3:

```
     ∇ Z←N TIMES A
[1]    Z←A
[2]    →(N=1)/0
[3]    Z←A+(N-1) TIMES A
     ∇

       5 TIMES 3.2                    4 TIMES 2 ¯1 0 1
16                              8 ¯4 0 4
```

10.9 Example: Generating Permutations

Consideration of the linear permutations of n distinct objects arises in various mathematical applications. The numerals $1, 2, \ldots, n$ can serve as a prototypical set of n objects; for example, the set of all permutations of three objects may be represented by {123, 132, 213, 231, 312, 321}. The search for a procedure for generating all permutations of n objects leads, rather naturally, to an inductive approach. For example, having all the permutations of 1 and 2—namely, {12, 21}, suggests that the permutations of 1, 2, and 3 be generated by using each of the two orderings of 1 and 2 together with the 3 in each of the three possible positions. An inductive approach, in turn, suggests a recursive function.

The recursive function *ALLPERMS* generates the permutations of the elements of ⍳*N*, giving the permutations as the rows of a matrix:

```
     ∇ Z←ALLPERMS N
[1]    →2×N>,Z←1 1 ⍴ 1                    ⍝ TEST IF N>1
[2]    Z←(⍳!N)⌽N≠N,ALLPERMS N-1
     ∇
```

The "workings" of this function are elucidated by study of the example execution *ALLPERMS* 3. The results of the steps in the right-to-left execution of line 2 are presented in succession:

```
      A←ALLPERMS 2
      A
1 2
2 1
```

Get all permutations on $N-1$ letters.

```
      B←3,A
      B
3 1 2
3 2 1
```

Attach N to all those permutations. This gives all permutations on N symbols with N in first position.

```
      C←3≠B
      C
3 1 2
3 1 2
3 1 2
3 2 1
3 2 1
3 2 1
```

Make N copies of each permutation in B.

```
      Z←(⍳6)⌽C
      Z
1 2 3
2 3 1
3 1 2
2 1 3
1 3 2
3 2 1
```

Rotate the N in each of those permutations into each of the N possible positions.

10.10 Example: Adaptive Integration

An adaptive technique for numerical integration is presented in this section. It provides an opportunity to introduce several functions that call on other functions, including one that is recursive. Elementary techniques for numerical integration such as composite Simpson's or trapezoidal rule work with uniform partitions of the interval of integration. By contrast, this adaptive method uses finer partitions in regions in which the integrand is more variable. The program presented in this section "adapts" itself so that no user oversight is required in selecting the nonuniform mesh.

The basic integrating function ***ROMS I MP*** (*ROM*berg, *SIM*pson) produces estimates of an integral based upon a five-point Simpson's rule ($n = 4$ subintervals) and a higher order Romberg five-point rule. The details of those rules can be found in numerical analysis texts such as Burden and Faires (1989). Notice, however, that the *INT*egration *Weigh**T***ing factors for those integration schemes are given below in the rows of the matrix ***I NTWT*** and that the second row has weights in the 1 4 2 4 1 ratio of composite Simpson's rule. The function ***ROMS I MP*** depends on the weights in the matrix ***I NTWT*** and the function ***F***, which is the integrand. Its arguments ***A*** and ***B*** give the endpoints of the interval of integration:

```
      INTWT←2 5 ⍴ 14 64 24 64 14 15 60 30 60 15 ÷ 180

    ∇ Z←A ROMS I MP B
[1]   Z←(B-A)×INTWT +.× F A+((B-A)÷4)×¯1+⍳5   ⍝ USES INTEGRAND F
    ∇
```

The function ***ROMS I MP*** is used to approximate the integrals:

$$\int_0^3 t^{2/3}\,dt \qquad\qquad \int_0^{\pi} \sin(t)\,dt$$

Notice that ***ROMS I MP*** results in two approximations of each of the definite integrals. The first number in the result is the Romberg-based approximation, which is usually better than the second number, which is the Simpson approximation. The integral of $\sin(t)$ has exact value 2:

```
    ∇ Z←F T
[1]   Z←T*2÷3
    ∇

      Z←0 ROMS I MP 3
      Z
3.721557892 3.71775948
      |-/Z
0.003798411642
```

```
    ∇ Z←F T
[1]   Z←1○T
    ∇

      Z←0 ROMS I MP ○1
      Z
1.998570732 2.004559755
      |-/Z
0.005989023161
```

The magnitude of the difference between the approximations given by ***ROMS I MP*** gives an estimate of the error in the approximation of the integral. Thus, ***ROMS I MP*** provides both an approximation for the integral and an easy way to estimate the accuracy of that approximation. It is desirable to have a method that provides much more accuracy than ***ROMS I MP***.

The goal is a technique that uses the estimate of the error provided via ***ROMS I MP*** to determine where more computation is required and where the approximation is satisfactory. The function ***A I RS*** (*A*daptive *I*ntegration: *R*omberg *S*impson) is defined as follows: If ***A I RS*** is applied to an interval $[a, b]$ and the error of ***ROMS I MP*** on that interval is within tolerance, then ***A I RS*** results in the ap-

proximation given by *ROMS IMP*. If there is too much error, the result is the sum of approximations of the integrals provided by *A IRS* on $[a, m]$ and $[m, b]$, where m is the midpoint of $[a, b]$. This definition of *A IRS* entails a natural recursion; see line 5.

Notice on line 1 of *A IRS* the result *Z* is assigned to be the result of *ROMS IMP* and the *EST*imated *ERR*or, *ESTERR*, is computed. Line 3 determines whether the error is acceptable; if not, line 5 results in the sum of approximations on the "bisected" interval. The error tolerance is also effectively divided in half: The approximation provided by *ROMS IMP* on each half-interval is acceptable only if it has error less than half the tolerable error on the whole interval.

It is convenient to control the error by considering the error per unit. The global variable *TOLERRPU* (*TOL*erable *ERR*or *P*er *U*nit) is multiplied by the magnitude of *B−A* to find the acceptable error on an interval of length *B−A*. That acceptable error is compared to the estimated error computed in line 1. The variable *NRS* used on line 2 counts the *N*umber of calls to *ROMS IMP* for later analysis; it is not used by the algorithm. Here is *A IRS*:

```
    ∇ Z←A AIRS B;M
[1]   ESTERR←|-/Z←A ROMSIMP B
[2]   NRS←NRS+1                    ⍝ COUNT CALLS TO ROMSIMP
[3]   →(ESTERR<TOLERRPU×|B-A)/0    ⍝ TEST IF ESTIMATE IS ACCEPTABLE
[4]   M←(A+B)÷2                    ⍝ IF NOT, FIND THE MIDPOINT
[5]   Z←(A AIRS M)+M AIRS B        ⍝ CALL AIRS ON HALF INTERVALS
    ∇
```

This function is now run on the two functions considered above. In the examples *TOLERRPU* is chosen so that the total tolerable error on the interval is $1E^{-}5$:

```
    ∇ Z←F T
[1]   Z←T*2÷3
    ∇

      NRS←0
      TOLERRPU←1E¯5÷3
      0 AIRS 3
3.744150642 3.744148676
      NRS
32
```

```
    ∇ Z←F T
[1]   Z←1○T
    ∇

      NRS←0
      TOLERRPU←1E¯5÷○1
      0 AIRS ○1
1.999999925 2.00000559
      NRS
11
```

Notice that the function $t^{2/3}$ required more calls to the integrator function. That is not surprising since it has unbounded derivatives near 0.

A user friendly version of this adaptive integration program is developed in Section 10.14.

10.11 Quad: Input and Output

In Section 3.12 the use of quad, ⎕, for output was introduced. Recall that assignments to quad result in the display of data in the session log in the order that the quads are encountered. The displayed data are passed on as a result that is available for further computation:

```
      3+⎕←2
2
5
```

```
      ⎕←A←2 2⍴⍳4
1 2
3 4
```

```
      (⎕←3+5)÷⎕←1+1
2
8
4
```

Quad may also be used to request user input; this is indicated by writing a quad in an expression wherever input is desired. The user is prompted for input; the user's response is evaluated and then used as an argument in place of the quad in the expression. For example,

```
      3+⎕              User types.
⎕:                     APL system requests input.
      5                User responds.
8                      Result of 3+⎕, with ⎕ replaced by 5.

      ⎕÷⎕              User types.
⎕:                     APL system requests input (right argument).
      3                User response is assigned to right quad.
⎕:                     APL system requests input (left argument).
      2+5              User responds with an expression.
2.333333333            Result of the requested quotient.

      'HI  ',⎕
⎕:
      'JEFF'           Input may be character data.
HI  JEFF

      ⎕÷⎕
⎕:
      ;;;6             Meaningless entry.
DOMAIN ERROR           An error message is produced.
      ;;;6
        ^
⎕:                     The user is reprompted.
      6                The right argument is entered.
⎕:
      12 6 18          The left argument for ÷.
2 1 3
```

The meaningless entry resulted in an error, and the APL system prompted again for input. The errors that will cause the user to be reprompted depend upon the particular APL system.

Quad input is used in an example in Section 10.13.

10.12 Quote-Quad: Input and Output

Quote-quad, ⍞, can be used for output that is not followed by a carriage return. It can also be used for literal input where the character nature of the input is understood so that quotes are not used to enclose the character vector. Assignments to quote-quad are catenated together "in the variable ⍞" until a call to quad or quote-quad has occurred or the display width is exceeded, at which time the "contents" of ⍞

are displayed. Here is an example where quad input causes the display of the contents of ⎕:

```
      ⎕←'PLEASE '
      ⎕←'ENTER THE NUMBER OF BOOKS REQUIRED:'
      N←⎕          ⍝CALL TO QUAD CAUSES DISPLAY OF ⎕ CONTENTS
PLEASE ENTER THE NUMBER OF BOOKS REQUIRED:
⎕:
      25
      N
25
```

When quote-quad input is the reason that display of the contents of ⎕ occurs, the user response is catenated to the displayed message and is available for use as the result of the quote-quad input. The user's response to the name request below is underlined so that it may be distinguished from the computer output. Notice that no quotes are used in the user response:

```
      ⎕←'PLEASE ENTER YOUR NAME: '
      N←⎕
PLEASE ENTER YOUR NAME: JOSHUA SMITH
      N
PLEASE ENTER YOUR NAME: JOSHUA SMITH
```

To extract the user input from *N*, the function drop (Section 8.1) is used. For example, `24↓N` discards the first 24 entries in the vector *N* leaving the user input:

```
      5↓'ABCDEFG'          3↓'ABCDEFG'          24↓N
FG                    DEFG                  JOSHUA SMITH
```

At first glance it may seem inconvenient to have the user's response catenated to the prompt message; however, this is quite handy for offering default options that may be modified by a respondent using editing keys:

```
      ⎕←'ENTER YOUR CLASS: CS101'
      CLASS←⎕
ENTER YOUR CLASS: CS101
```

The user backspaces twice and types 52. The modified line is as follows:

```
ENTER YOUR CLASS: CS152
      CLASS
ENTER YOUR CLASS: CS152
      18↓CLASS
CS152
```

Quote-quad input is used in programs in which the program prompts for character input. Exiting a program that interminably requests character input is accomplished with a strong interrupt; see Section 3.7.

10.13 Example: Entering a Matrix Name-List

Consider a simple utility function, *NAMES I N*, for entering a character matrix without user concern about the length of the rows or needs for enclosing the character data in quotes. This might be used for entering a list of names with one name in each row. Throughout execution *Z*, *M*, and *N* are maintained so that the shape of *Z* is *M*, *N*. On line 4, the partial result *Z* and the newest row, *ROW*, which is the result of ⍞ input, are extended to compatible lengths before being catenated along the first axis:

```
     ∇ Z←NAMES I N;M;N;MAXM;ROW
[1]    ⎕←'PLEASE ENTER THE NUMBER OF ROWS:'
[2]    MAXM←⎕ ◊ M←N←0 ◊ Z←0 0⍴' '
[3]    ⎕←'PLEASE ENTER ROWS ONE AT A TIME:'
[4]  MORE:ROW←⍞ ◊ N←N⌈⍴ROW ◊ Z←((M,N)↑Z),[1](1,N)⍴N↑ROW
[5]    →MORE×MAXM>M←M+1
     ∇

      NAMES←NAMES I N
PLEASE ENTER THE NUMBER OF ROWS:
⎕:
      6
PLEASE ENTER ROWS ONE AT A TIME:
JOHN                                    User inputs six rows.
SUSAN
CHARLES
WINNIE
LINDA
CHRISTY

      NAMES                                   '*',NAMES,'*'
JOHN                                  *JOHN   *
SUSAN                                 *SUSAN  *
CHARLES                               *CHARLES*
WINNIE                                *WINNIE *
LINDA                                 *LINDA  *
CHRISTY                               *CHRISTY*

      ⍴NAMES
6 7
```

A version of this program with modified input and output styles is considered in Section 10.18. Both versions employ a matrix representation of name lists. Some systems provide enclosed or nested arrays that allow name lists to be represented as a vector of words. The present program, however, is valuable for inputting character matrices in general.

10.14 Example: More Adaptive Integration

The *ADAP*tive *INT*egration program, *ADAP I NT*, initializes the variables used by *A I RS* and prompts the user for input. The program results in global variables: *TOLERRPU* gives the tolerable

error per unit step in terms of the total desired tolerance the user provides, and *ZAI* holds the approximations to the integral. The program also produces a report of the Romberg approximation for the integral and the number of calls to *ROMSIMP* used:

```
     ∇ ADAPINT;A;B;INT
[1]    ⎕←'WARNING: THE INTEGRAND IS ASSUMED TO BE THE FUNCTION F'
[2]    ⎕←'PLEASE ENTER THE INTERVAL OF INTEGRATION AS A VECTOR:'
[3]    INT←⎕ ⋄ A←INT[1] ⋄ B←INT[2]
[4]    ⎕←'PLEASE ENTER THE DESIRED TOLERANCE:'
[5]    TOLERRPU←|⎕÷B-A
[6]    NRS←0                ⍝ NRS IS THE NUMBER OF CALLS TO ROMSIMP
[7]    ZAI←A AIRS B         ⍝ ZAI CONTAINS ADAPTIVE APPROXIMATIONS
[8]    ⎕←'THE ESTIMATED INTEGRAL IS:'
[9]    ⎕←ZAI[1]
[10]   ⎕←'THE NUMBER OF CALLS TO THE INTEGRATOR WAS:'
[11]   ⎕←NRS
     ∇
```

Consider an example in which the integrand is $F(t) = t^{2/3}$:

```
      ADAPINT
WARNING: THE INTEGRAND IS ASSUMED TO BE THE FUNCTION F
PLEASE ENTER THE INTERVAL OF INTEGRATION AS A VECTOR:
⎕:
      0 3
PLEASE ENTER THE DESIRED TOLERANCE:
⎕:
      1E¯5
THE ESTIMATED INTEGRAL IS:
3.744150642
THE NUMBER OF CALLS TO THE INTEGRATOR WAS:
32
      ZAI
3.744150642 3.744148676
```

One of the surprising aspects of programming that *ADAPINT* illustrates is that user prompts, variable initiation, and organization of output can take far more program space than the underlying computation. The function *ADAPINT* is far longer than the functions used for the computations.

10.15 Format

The monadic function *format*, denoted ⍕, results in a character array that has the same appearance as the displayed result of its right argument. Format is useful for making tables easier to read, merging text and data, and sharing information with non-APL systems. Notice that ⍕ produces no surprises in appearance, but that the shapes requested in the examples below provide a clue that characters are being counted. For example, the fact that ⍴⍕⍳6 is 11 indicates that the character vector ⍕⍳6 consists

of the character 1, space, the character 2, space, ..., the character 6:

```
      ⍕⍳6
1 2 3 4 5 6

      ⍕2 3⍴7
7 7 7
7 7 7

      ⍕'MARIBETH'
MARIBETH

      ⍴⍕⍳6
11

      ⍴⍕2 3⍴7
2 5

      ⍴⍕'MARIBETH'
8

      ⍕-÷3
¯0.3333333333

      ⍕112233445566
1.122334455E11

      'N = ',⍕25
N = 25

      ⍴⍕-÷3
13

      ⍴⍕112233445566
14

      ⍴'N = ',⍕25
6
```

The appearances do depend upon system variables such as ⎕*PP*, and the exact shape of the result does vary to some extent from system to system. Notice that the last example allows both text and a numeric result to be displayed adjacently. Some other examples of that type are

```
      A←2 5⍴⍳10
      (2 4⍴'A =   '),⍕A
A = 1  2  3  4  5
    6  7  8  9 10

      U← 1 3 4 6 7
      'THE AVERAGE OF THE ',(⍕⍴U),' ENTRIES OF U IS:',⍕(+/U)÷⍴U
THE AVERAGE OF THE 5 ENTRIES OF U IS: 4.2
```

The format function can also take a left argument. The left argument contains pairs of integers that specify the field widths and number of decimal places to be displayed. If only one pair is given, all columns are formatted accordingly; otherwise, a pair of integers is required for each column:

```
      A←3 4 5∘.÷⍳5
      A
3      1.5      1      0.75      0.6
4      2        1.333  1         0.8
5      2.5      1.666  1.25      1

      5 2⍕A
3.00 1.50 1.00 0.75 0.60
4.00 2.00 1.33 1.00 0.80
5.00 2.50 1.67 1.25 1.00

      3 0 5 1 6 2 6 2 5 1 ⍕A
  3  1.5  1.00  0.75  0.6
  4  2.0  1.33  1.00  0.8
  5  2.5  1.67  1.25  1.0

      3 0 5 1 8 4 6 2 5 1 ⍕A
  3  1.5  1.0000  0.75  0.6
  4  2.0  1.3333  1.00  0.8
  5  2.5  1.6667  1.25  1.0
```

Many systems have a related system function ⎕*FMT* (see Appendix B). Typical kinds of features allow the user to specify the width and number of decimal places for each column of matrix output, to format by an example row, to specify exponential or floating point formatting, to embed the data into given character decorator fields, to replace characters, such as minus for high minus, to fill in trailing zeros, and to insert commas for every third digit to the left of a decimal point. Features vary from system to system; see your system documentation.

10.16 Example: Formatting an Interest Table

Consider the "formatting" problem of inserting blanks into a large table of entries so that horizontal lines can followed. Here is an interest table for interest rates 6%, 8%, and 10% compounded annually for 1 to 15 years. Let ⎕*PP*←5. The blank rows are inserted using expansion (see Section 8.5) and a heading is added:

```
      A←(15 1⍴⍳15),⍉ 1.06 1.08 1.1 ∘.* ⍳15
      B←(17 ⍴ 1 1 1 1 1 0)⌿⍪A
      C←'YEAR        6%       8%       10% ',[1]'-',[1]B

      A
 1       1.06     1.08     1.1
 2       1.1236   1.1664   1.21
 3       1.191    1.2597   1.331
 4       1.2625   1.3605   1.4641
 5       1.3382   1.4693   1.6105
 6       1.4185   1.5869   1.7716
 7       1.5036   1.7138   1.9487
 8       1.5938   1.8509   2.1436
 9       1.6895   1.999    2.3579
10       1.7908   2.1589   2.5937
11       1.8983   2.3316   2.8531
12       2.0122   2.5182   3.1384
13       2.1329   2.7196   3.4523
14       2.2609   2.9372   3.7975
15       2.3966   3.1722   4.1772

      B
 1       1.06     1.08     1.1
 2       1.1236   1.1664   1.21
 3       1.191    1.2597   1.331
 4       1.2625   1.3605   1.4641
 5       1.3382   1.4693   1.6105

 6       1.4185   1.5869   1.7716
 7       1.5036   1.7138   1.9487
 8       1.5938   1.8509   2.1436
 9       1.6895   1.999    2.3579
```

```
10        1.7908   2.1589   2.5937

11        1.8983   2.3316   2.8531
12        2.0122   2.5182   3.1384
13        2.1329   2.7196   3.4523
14        2.2609   2.9372   3.7975
15        2.3966   3.1722   4.1772
```

```
          C
YEAR        6%       8%       10%
──────────────────────────────────
 1        1.06     1.08     1.1
 2        1.1236   1.1664   1.21
 3        1.191    1.2597   1.331
 4        1.2625   1.3605   1.4641
 5        1.3382   1.4693   1.6105

 6        1.4185   1.5869   1.7716
 7        1.5036   1.7138   1.9487
 8        1.5938   1.8509   2.1436
 9        1.6895   1.999    2.3579
10        1.7908   2.1589   2.5937

11        1.8983   2.3316   2.8531
12        2.0122   2.5182   3.1384
13        2.1329   2.7196   3.4523
14        2.2609   2.9372   3.7975
15        2.3966   3.1722   4.1772
```

10.17 Execute

The monadic function *execute*, denoted ⍎, is used to interpret character vectors as APL expressions. Thus, ⍎ '2*8' means execute the three character string '2*8' as an APL expression; the result is the number 256. This function is valuable in many ways. It allows numeric interpretation of character representations of data, it allows programs to use APL functions or variables whose names are provided by quote-quad input, and it allows execution of an expression created within a function:

```
      '3+5'                  1+⍎'3.14'              ⍎'A←⍳5'
3+5                    4.14                                  (No display.)

      ⍎'3+5'                 ⍎7⍴'2+'                A
8                      8                        1 2 3 4 5

      ⍎'3*¯2'                20<⍎'23 ¯23 50'        B×⍎'B←⍳3'
0.1111111111           1 0 1                    1 4 9
```

10.18 Example: Entering a Matrix Name-List, Revisited

Consider the program *NAMES I N2*, which is a modification of the function *NAMES I N* given in Section 10.13. It has slightly cleaner input and output and prompts for the name of the matrix it creates.

On lines 2 and 4, the drops separate the ⍞ prompt used on the previous lines from the user response. Note the use of execute on line 4, which gives a numerical value indicated by a character string. On line 8, execute assigns the result of the input to the name previously indicated by the user. Here is *NAMES I N2*:

```
     ∇ NAMES I N2 ; NAM ; M ; N ; MAXM ; ROW ; Z
[1]    ⍞←'ENTER A NAME FOR THE MATRIX:
[2]    NAM←29↓⍞
[3]    ⍞←'ENTER THE NUMBER OF ROWS: '
[4]    MAXM←⍎26↓⍞
[5]    ⎕←'ENTER THE ROWS:' ◊ M←N←0 ◊ Z←0 0⍴' '
[6]  MORE:ROW←⍞ ◊ N←N⌈⍴ROW ◊ Z←((M,N)↑Z),[1](1,N)⍴N↑ROW
[7]    →(MAXM>M←M+1)/MORE
[8]    ⍎NAM,'←Z'
     ∇
```

```
      NAMES I N2
ENTER A NAME FOR THE MATRIX: MAT1      User response underlined.
ENTER THE NUMBER OF ROWS: 3            User response underlined.
ENTER THE ROWS:
ALPHA                                  User enters three lines.
BETA
GAMMA

      MAT1                             Check the matrix.
ALPHA
BETA
GAMMA
```

10.19 Example: Matrix to a Power-of-2 Power

The execute function provides a simple way to compute high powers of a matrix using repeated squaring, without any branching or loops. First notice,

A^2	is	`A+.×A`	Which is	`⍎ 5⍴'A+.×A←'`
A^4	is	`A+.×A←A+.×A`	Which is	`⍎11⍴'A+.×A←'`
A^8	is	`A+.×A←A+.×A←A+.×A`	Which is	`⍎17⍴'A+.×A←'`

A^{2^N} is the result of executing `(5+6×N-1)⍴'A+.×A←'`

An APL function to compute these *E*xponential *POW*ers is *EPOW*:

```
     ∇ Z←A EPOW N
[1]    Z←⍕(5+6×N-1)⍴'A+.×A←'
     ∇
```

```
      A
1 2 3
4 5 6
7 8 9
      A EPOW 1
 30  36  42
 66  81  96
102 126 150
      A EPOW 2
 7560  9288 11016
17118 21033 24948
26676 32778 38880
```

```
      B
0.1 0.6 0.3
0.3 0.4 0.3
0.2 0.6 0.2
      B EPOW 1
0.25 0.48 0.27
0.21 0.52 0.27
0.24 0.48 0.28
      B EPOW 4
0.2272727273  0.5  0.2727272727
0.2272727273  0.5  0.2727272727
0.2272727273  0.5  0.2727272727
```

Notice the entries of the 4th power of *A* are huge, whereas the 16th power of the Markov matrix *B* has constant rows of modest magnitude.

10.20 Functions as Arguments of Functions

It is sometimes convenient to have a function serve as an explicit argument to a function. Execute provides an easy way for calling a function given as an argument. Previously, functions were called by a specific name and were not explicit arguments of the calling function.

Consider the problem of computing the slope of a secant line, m_{sec}, through two given points, $(a,f(a))$ and $(b,f(b))$, on the graph of a function $f(x)$:

$$m_{sec} = \frac{f(a) - f(b)}{a - b}$$

Here the problem of having the function f as an argument to be specified at execution time is considered. Two solutions are presented.

First, consider the solution when the function is addressed by its name. Suppose functions *F*, *LN*, and *COSH* are already defined in the workspace. The function *MSEC* takes a function name given as a character vector as its left argument and the two points of evaluation as its right argument. Execute is applied to the function name catenated to a string giving the name of the variable containing the points. Some examples are computed:

```
     ∇ Z←F T
[1]    Z←1○T
     ∇
```

```
     ∇ Z←LN T
[1]    Z←⍟T
     ∇
```

```
     ∇ Z←COSH X
[1]    Z←0.5×(*X)+*-X
     ∇
```

```
     ∇ Z←FN MSEC X
[1]    Z←(-/⍎FN,' X')÷-/X
     ∇
```

```
      'F' MSEC 0 .1
0.9983341665

      'F' MSEC 0 .01
0.9999833334

      'LN' MSEC 1 1.1
0.953101798

      'COSH' MSEC 0 .1
0.05004168056
```

The second solution involves taking an expression describing the function as left argument. By convention the independent variable is assumed to be *X*. Execute can now be applied directly to the expression describing the function. Some examples are given:

```
     ∇ Z←EXPR MSEC X
[1]    Z←(-/⍎EXPR)÷-/X
     ∇

      'X*2' MSEC 0 1
1

      '1○X' MSEC 0 .1
0.9983341665

      '(1+X)÷1-X' MSEC 0 .1
2.222222222
```

Exercises

1. Write definitions of monadic functions that take a vector argument and return a result that compute the (a) arithmetic mean, (b) geometric mean, (c) harmonic mean of a list of positive numbers.
2. Write a program *MEANS* that has a vector argument, does not pass on a result, and calls on the three functions of the previous exercise, displaying the results together with appropriate names.
3. Compare the output of the following version of *STATISTICS* with that given in Example 10.2; use the same vector *V*:

```
     ∇ STATISTICS V
[1]    2 1⍴' '
[2]    '                RANGE: ',⍕ RANGE V
[3]    '             MODE(S): ',⍕ MODE ⌊.5+V
[4]    '               MEDIAN: ',⍕ MEDIAN V
[5]    '              AVERAGE: ',⍕ AVG V
[6]    'STANDARD DEVIATION: ',⍕ SD V
[7]    '  '
[8]    'HISTOGRAM:'
[9]    HIST ⌊.5+V
     ∇
```

4. (Continuation of Example 10.5.) The Borda alternative is the alternative (if there is a unique one) that receives the lowest total sum of the preference rankings assigned by the voters. Write a monadic function with argument *B* (the Ballot) that determines the Borda alternative or indicates that there is none. Try your function on Ballots *B* and *C* in Example 10.5.
5. (Continuation of Example 10.5.) In the theory of voting the Copeland count for an alternative is the difference between the number of wins and the number of losses in pairwise contests of that alternative with each of the other alternatives [see Straffin (1980)]. Write a monadic function *COPELAND* with argument *B* (the Ballot) that produces the vector of Copeland counts for the alternatives on the Ballot. Apply your function to ballots *B* and *C* in Example 10.5. (Hint: Use *B*+.<⍉*B*.)
6. Write a monadic function *COPEWIN* that uses *COPELAND* (Exercise 5) to determine the unique alternative with highest Copeland count or indicates there is none.
7. Write a function *REMOVEKS* that takes a positive integer *N* as the argument and returns the largest factor of *N* that has no factor of *K* in it. See *REMOVE2S* in Section 10.4.
8. Write alternative versions of (a) *MAJORITY*, (b) *PLURALITY*, and (c) *BORDA* that use no branches. See Section 10.5 and Exercise 4.
9. Write an alternative version of *REMOVE2S* (Section 10.3) that does not use branching.
10. Write an alternative version of *REMOVEKS* that does not use branching. See Exercise 7.
11. Rewrite *TERNEWT* in Section 3.10 with line labels and branches using compression.
12. Rewrite the three-way branch in line 3 of *BISECTMETHOD* in Section 10.7, replacing it with a branch determined by indexing a vector of line labels.
13. Compare the effects of

    (a) →(*N*<10)/*LINELABEL*
    (b) →(*N*<10)⍴*LINELABEL*
    (c) →(*N*<10)↑*LINELABEL*
    (d) →*LINELABEL*×⍳(*N*<10)

14. Describe the effect of running the niladic function

```
     ∇ ROLLEM
[1]    → ≠/□←?6 6
     ∇
```

15. Observe the results from *TRANS* when *V* is *any* four-element numeric vector. Then explain what the function does.

```
     ∇ TRANS V
[1]    →∨/0≠□←V←|V-1⌽V
     ∇
```

    For further information see Magyar.
16. Write a function *THREE1TWO* that transforms a positive integer *N* by the following procedure: Triple it, add 1 to that, then remove all factors of 2 from that result (e.g., 7 yields 21, 22, and finally 11 by this scheme). Repeat the application of this procedure until the number 1 occurs or the scheme leads to a repeating cycle of numbers. Thus, starting with 7 would lead to the sequence: 7, 11, 17, 13, 5, 1, at which point execution should stop. The function should display the sequence generated.
17. Predict the result of the following expressions when followed by the indicated input. Check your results.

    (a) □+□ followed by 3 and 5 6 7
    (b) □+□←□ followed by 3 and 5 6 7
    (c) □+□←□+□ followed by 3 and 5 6 7 and ¯5

18. Write a program *ROWOP* for accomplishing elementary row operations on a matrix. The program should have a matrix *A* as its right argument and prompt the user to select one of the three elementary row operations (or to exit). It should also prompt for scale factors and row indices as appropriate. The modified matrix should be displayed and further changes be possible. The final modified matrix should be returned as a result.

19. Predict the result of the following expressions when followed by the indicated input. Check your results.

(a) `⎕←'HERE' ◊ A←⎕` followed by *I S SAM*. What is *A* and `⍴A`?
(b) `⎕←'HERE' ◊ B←⎕` followed by `'I S SAM'`. What is *B* and `⍴B`?

20. Use *ADAP I NT* from Section 10.14 to approximate the following integrals with total tolerances (i) `1E¯5` (ii) `1E¯10`.

(a) $\int_0^1 \frac{4}{t^2+1}\,dt$ **(b)** $\int_1^8 \frac{1}{t}\,dt$

(c) $\int_0^4 \sqrt{t}\,dt$ **(d)** $\int_0^1 \sin^{-1}(t)\,dt$

21. Write a function *ADAP I NT2* that is a modification of *ADAP I NT* in Section 10.14 so that the user is prompted for a maximum number of calls to *ROMS I MP*, and *ADAP I NT2* halts with a message if the maximum number of calls is exceeded. Test your answer on the problems in Exercise 20 with 100 as the maximum number of calls to *ROMS I MP*.

22. Write a function *ADAP I NT3* (a modification of *ADAP I NT* in Section 10.14) that creates the matrix global variable *I NTA I* that contains as rows the intervals used by *ADAP I NT*. The intervals should be ordered when the program finishes. Test your answer on the problems from Exercise 20.

23. Predict the effect of the following expressions. Check your work.

(a) `⍕2 2⍴⍳6` **(b)** `'*',(⍕2 2⍴⍳6),'*'` **(c)** `'N = ',(⍕⍳5),' ETC.'`

24. Write a function *SK I PROW* that inserts a blank row after every *K* rows of the matrix *A*, where *K* is the left argument and *A* is the right argument. The function should handle both arguments that are numeric or arguments that are character.

25. Modify the formatting of the interest table matrix *A* given in Section 10.16 so that there are only two blanks between the first and second columns. Include trailing zeros.

26. Write a function *I NTERESTTABLE* that takes as its left argument a number of years and takes as a right argument a vector of percentages. It should yield an interest table resulting from annual compounding for the years up to the indicated year and for the given percentages. Have the result skip a line after each 5 years and display data to 5 significant figures. Head the rows by the year number and the columns by the percentages. Test your program using 20 years and 6, 8, 7.5, and 11%.

27. Write a function *L I NECOL* that takes a numeric matrix as its only argument and results in a character matrix with vertical bars inserted on the left and right side of each column. For example,

```
      A
1  2  2.286
3  4  1.234
5  6  3.512
```

```
      LINECOL A
| 1 | 2 | 2.286 |
| 3 | 4 | 1.234 |
| 5 | 6 | 3.512 |
```

Test *L I NECOL* on *A* and on `(⍳5)∘.÷⍳6`.

28. Predict the effect of the following expressions. Check your work.

(a) `⍎5⍴'3+'` **(b)** `⍎(5⍴'A'),'←2 2⍴⍳4'` **(c)** `⍎□←5⍴'3+'`

29. Use the program in Section 10.19 to find the fourth power of the matrices below. Observe whether the high powers have large entries and seem to have uniform rows.

(a)
```
1 1 1
1 1 1
1 1 1
```

(b)
```
 1 ¯1  1
¯1  1 ¯1
 1 ¯1  1
```

(c)
```
0.6 0.2 0.2
0.1 0.4 0.5
0.3 0.3 0.4
```

(d)
```
0.1 0.2 0.6
0.3 0.5 0.2
0.2 0.3 0.2
```

(e)
```
1 0 1
0 1 0
1 0 1
```

(f)
```
  0.25   1.5   1.25
  0.25   0    ¯0.25
¯0.25  ¯0.5   ¯0.25
```

30. Write a function *MPOW* that computes the *N*th power of a matrix. The left argument should be the matrix and the right argument should be the power. Avoid using a loop, branches, or recursion. See Section 10.19.

31. Run *MPOW* to compute the 7th, 15th, and 16th powers of the matrices from Exercise 29.

32. Predict the result of the following expressions. Check your work.

(a)
```
U←6-⍳7
⍒U
⍎⍒U
U∧.=⍎⍒U
```

(b)
```
V←'5 6 7'
⍎V
⍒⍎V
V∧.=⍒⍎V
```

Appendix A Help, Error Messages, and Debugging

Appendix A contains several kinds of help for APL users. It includes quick resolutions of unexpected behavior, a discussion of APL error messages, and use of suspended functions for debugging.

A.1 Surprises for New Users

Here are some difficulties commonly encountered by new APL users, as well as their resolutions:

Symptom	*Remarks*
[3] as a prompt	System repeatedly prompts with a number in square brackets. This means function editing mode has been entered, possibly inadvertently. Use ∇ to close definition mode. See Section 3.1.
⎕: as a prompt	The system is awaiting numeric input. This often occurs as an error when the del is left off of a request to display a function definition. The simplest response is to enter a number. Alternately, a strong break, ⟨ctrl-break⟩–⟨ctrl-break⟩, can be used. See Section 3.7.
No response to input	This may mean that the system is gathering character data from the keyboard (in response to ⍞). A strong break (⟨ctrl-break⟩–⟨ctrl-break⟩) can be used to interrupt the input process. See Section 3.7. Some systems exhibit this behavior if a quotation mark around character data is not closed with a second quotation mark. Those systems are "gathering" the carriage returns as part of literal input.
Variable disappeared	A user-entered variable seems to disappear without the user erasing it. A variation on this problem occurs when the variable value is changed. First, look at) ***VARS*** to check that the variable name was not misspelled. Another possibility is that a program that has a global variable with the same name has been run. Check any programs that were run. A more subtle reason for the problem occurs when a suspended function has a local variable with the same name. In that case, clear suspended functions with) ***RESET***. See Sections 3.7 and A.3.

A.2 Error Messages

Error messages provide a clue into the reason a difficulty is encountered. Most error messages are descriptive enough to be helpful. This section describes some frequently occurring errors.

A *syntax error* is an error in the structure of an expression. These occur when a requested computation does not describe a meaningful order of computation. For example, parentheses may be mismatched or a function may not have the correct number of arguments. Notice the system indicates the error, retypes the line, possibly with some spacing changes, and places a caret under the position where the error was recognized. Here are examples:

```
      8)+2
SYNTAX ERROR
      8)+2
        ^
```
The parenthesis is not matched.

```
      3×12+
SYNTAX ERROR
      3×12+
          ^
```
The addition sign has no right argument.

```
      ¯A←1+2
SYNTAX ERROR
      ¯A←1+2
      ^
```
High minus is not a function.

```
      A
3
```
A was assigned a value by the previous flawed expression.

```
      4A
SYNTAX ERROR
      4 A
        ^
```
Juxtaposition is not a function.

The example `¯A←1+2` is interesting because an error occurred but the computation was accomplished to the point where the assignment appeared.

Reference to an undefined variable or function results in a *value error*. For example,

```
      )CLEAR
```
No user-assigned variables or functions are in the workspace.

```
      3+A
VALUE ERROR
      3+A
        ^
```

```
      F 4 5 6
VALUE ERROR
      F 4 5 6
      ^
```

An error that occurs when a computation is requested on data outside the domain of the function results in a *domain error*. For example, an attempt to divide by 0 or take the logarithm of 0 results in a domain error. Some systems produce domain errors when the square root of a negative number is requested; other systems are extended to provide a complex root. Examples are

```
      ÷0                          Reciprocal of 0 is not defined.
DOMAIN ERROR
      ÷0
      ∧

      3 4 1÷1 2 0                 1÷0 is not defined.
DOMAIN ERROR
      3 4 1 ÷ 1 2 0
            ∧

      0÷0                         APL does assign a value to 0÷0.
1

      ⍟0                          The logarithm of 0 is not defined.
DOMAIN ERROR
      ⍟0
      ∧

      (X←1)÷Y←0                   Notice both arguments are successfully
DOMAIN ERROR                      computed before the error occurs.
      (X←1)÷Y←0
      ∧

      X
1
      Y
0

      'A'+'B'                     Addition is not defined for characters.
DOMAIN ERROR
      'A'+'B'
        ∧
```

An *index error* occurs when an array is indexed outside the meaningful range of indices. For example,

```
      A
2  4  6  8 10 12
      ⍴A
6
      A[4]
8
      A[7]
INDEX ERROR
      A[7]
      ∧
```

```
      B
1 2 3 4
5 6 7 8
      B[2;1 3]
5 7
      B[1;5]
INDEX ERROR
      B[1;5]
      ∧
```

```
      A[¯2 3]
INDEX ERROR
      A[¯2 3]
      ∧
```

Many APL functions require that the number of entries along an axis match. For example, two vectors can be added, `2 3 8+7 9 4`, but the number of entries on each side must match (unless one is a single element that is extended to match the length of the other side). When the number of components on each side differs, there is a *length error*. A length error is an indication that the APL function used requires the length along some axes to match but that the axes do not match length for the given arguments. For example,

```
      2 3 4+6 1
LENGTH ERROR
      2 3 4 + 6 1
            ∧
```

Pairs cannot be added to triples.

```
      A←3 3⍴⍳9
      B←3 2⍴1 2 0
```

```
      A+.×B
 7  4
16 13
25 22
```

The product of a 3-by-3 with a 3-by-2 matrix is meaningful. See Section 4.1 for the matrix product.

```
      B+.×A
LENGTH ERROR
      B+.×A
      ∧
```

The product of a 3-by-3 with a 2-by-3 matrix is not meaningful.

```
      1 2 3↑A
LENGTH ERROR
      1 2 3 ↑A
            ∧
```

Take on a matrix requires a two-component left argument. See Section 8.3.

Sometimes an APL expression produces an error because the computation requested requires that the arguments have certain dimensions. For example, a vector cannot be added to a matrix and one cannot seek the index of data in a reference matrix (it is only possible to seek the index in a reference vector; see Section 9.4). Failure to meet such dimensionality conditions produces a *rank error*. Examples are

```
      A
1 2 3
4 5 6
7 8 9
      A+1 2 3
RANK ERROR
      A+ 1 2 3
       ∧
```

A matrix cannot be added to a vector.

```
      1 5 3⍳A          The indices of A in 1 5 3 can be found.
1 4 3
4 2 4
4 4 4

      A⍳1 5 3          The indices of 1 5 3 in A is not a
RANK ERROR             meaningful request.
      A⍳ 1 5 3
      ∧
```

Large computations can exceed the available computer memory. Such situations result in a *workspace full* error message. Of course, this error depends upon the computer hardware, the amount of memory allocated to APL, and the amount of space used by other functions and data in the workspace. Examples are

```
      1000000⍴2 3÷7
WS FULL
      1000000⍴2 3÷7
               ∧

      A←(?1000⍴100)∘.*¯1+⍳101
WS FULL
      A←(?1000⍴100)∘.*¯1+⍳101
        ∧
```

A *result error* occurs when a result required by a function is not available. This can happen when a function that does not return a result is part of a computation. Consider the function *FIBS*, which outputs the first *N* Fibonacci numbers but does not return a result; that is, the results are not available for further computation. See Section 10.1 for a discussion of functions that do not return a result. An example using *FIBS* follows:

```
    ∇ FIBS N;K;OLDZ;NEWZ;Z
[1]   OLDZ←0 ◊ ⎕←Z←1 ◊ K←1
[2]   →3×K<N
[3]   NEWZ←Z+OLDZ
[4]   OLDZ←Z ◊ ⎕←Z←NEWZ ◊ K←K+1
[5]   →2
    ∇

      FIBS 8
1
1
2
3
5
8
13
21
```

```
      ρFIBS 8
1
1
2
3
5
8
13
21
RESULT ERROR
      ρFIBS 8
        ^
```

Computations that are well posed might not run successfully because of space limitations or because execution time is excessive. Such a computation can be interrupted with the use of a strong or a weak interrupt. These are effected with one or two taps of the attention or the ⟨ctrl-break⟩ key (see Section 3.7):

```
      ⍳10000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
      25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
      44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62
      63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81
      .
      .      (Some details are omitted.)
      .
      1708 1709 1710 1711 1712 1713 1714 1715 1716 1717 1718
      1719 1720 1721 1722 1723 1724 1725 1726 1727 1728 1729
           (You strike the ⟨control-break⟩ key.)
      1730 1731 1732 1733 1734 1735 1736 1737 1738 1739 1740
```

Output halts after you strike the interrupt key.

A common and serious reason to interrupt a computation is that it is stuck in an infinite loop or it is taking far too long. An example of a function requiring such an interrupt might be the *FIBS* function given above run on a large problem:

```
      FIBS 1000
1
1
2
3
5
8
13
.
.                    Many details are deleted.
.
7.577916187E32
1.226132595E33
```

```
                    The ⟨ctrl-break⟩ key is struck.
FIBS[3]
```

This interrupt leaves the function in a suspended state that should be cleared with `)RESET` (see Section A.3).

Note that the error message a particular "bad" computation produces may be difficult to determine. For example, should `'ABC'+'DE'` be a domain error because addition is not defined for character data or should it be a length error because the addition arguments are not the same length? Indeed, which error results varies from system to system. Most of the errors that have been described here are put into the event reporting system variable `⎕ER` (see Appendix B).

A.3 Suspended Functions

When an error is encountered during execution of a function, APL "suspends" the function and issues an appropriate error message. The following is an example with `RHEP2` and `F` from Section 3.3. The integrand is taken to be $f(t) = 1/t$. The examples include a case in which the `X` values contain the entry `0` causing a domain error. The command `)SI` requests display of the state indicator:

```
     ∇ Z←N RHEP2 INT;A;B;DX;X;Y
[1]    A←INT[1]
[2]    B←INT[2]
[3]    DX←(B-A)÷N
[4]    X←A+DX×⍳N
[5]    Y←F X
[6]    Z←DX×+/Y
     ∇

     ∇ Z←F T
[1]    Z←1÷T
     ∇

       )VARS                    No variables in the workspace.

       8 RHEP2 1 2              Approximate some integrals.
0.6628718504
       8 RHEP2 ¯2 ¯1
¯0.7253718504
       8 RHEP2 0 2              Try an improper integral.
2.717857143
       8 RHEP2 ¯2 0             Function evaluation at 0 causes error.
DOMAIN ERROR
F[1]   Z←1÷T
         ∧
       )SI                      View the state indicator.
F[1]   *
RHEP2[5]
```

The state indicator shows that the function *F* was *suspended* on line 1. The asterisk indicates the fact that *F* is suspended. The function *RHEP2* called *F* on line 5. The function *RHEP2* is said to be *pendant*.

Notice in the next example that the variables of *F* and *RHEP2* are available for display and manipulation. This environment can be especially helpful for debugging a program. Here the fact that *X* (inside *RHEP2*) and *T* (inside *F*) contain 0 allows you to see the division by 0 problem quickly. Some confusion can arise when several global variables and the local variables of functions called are available. The system command) *SIV* gives the variables of the suspended and pendant functions:

```
      )VARS
A       B       DX      INT     N       T       X
      A
¯2
      X
¯1.75 ¯1.5 ¯1.25 ¯1 ¯0.75 ¯0.5 ¯0.25 0
      T
¯1.75 ¯1.5 ¯1.25 ¯1 ¯0.75 ¯0.5 ¯0.25 0
      Z
VALUE ERROR
      Z
      ∧
      )SIV     Or )SINL on some systems.
F[1]  * Z        T
RHEP2[5]         Z       N       INT     A       B       DX
X       Y
```

Even though *F* is the suspended function the variables *A*, *B*, and so on, from *RHEP2* are available in the workspace. This is evidence of the fact that variables local to a function, like *RHEP2*, are available to the functions that are called internal to that function. If variables of the same name appear in the workspace, they are not available until the suspension is cleared. Thus, it is important to clear the state indicator promptly.

At this point, you may choose to clear the state indicator with)*RESET* or make some modifications and try to complete the execution. In the example at hand, the most likely way to continue would be to reset the state indicator and do a more appropriate computation. It is possible, however, to change the 0 entry in *T* and continue. For example, setting *T*[8]←.0000001, you continue the execution with the interactive branch statement →1, which indicates that line 1 of the suspended function *F* should be recomputed and processing of functions pendant on that result should continue:

```
      T[8]←0.0000001
      T
¯1.75 ¯1.5 ¯1.25 -1 ¯0.75 ¯0.5 ¯0.25 0.0000001
      →1
2499997.407
```

Notice that the result is not very useful. Care should be used in restarting computations this way; for example, part of a function line may have incremented a counter before the error was evoked—restarting the computation with that line may inappropriately increment the counter again.

You can save a workspace containing suspended and pendant functions and investigate the situation later. It is possible for several functions to be suspended at the same time. The most recent sus-

pension is at the "top" of the state indicator. It is sometimes useful to clear the suspended functions one at a time. The niladic (no argument) branch resets only the top suspension. For example,

```
      8 RHEP2 ¯2 0          Error 1: RHEP2 calling F at 0.
DOMAIN ERROR
F[1]    Z←1÷T
           ∧

      F 0 9                 Error 2: F at 0.
DOMAIN ERROR
F[1]    Z←1÷T
           ∧

      8 RHEP2 1 2           The suspensions above do not interfere
0.6628718504                with proper computations.

      16 RHEP2 ¯10 0        Error 3: RHEP2 calling F at 0.
DOMAIN ERROR
F[1]    Z←1÷T
           ∧

      )SI                   Observe the three suspensions.
F[1]  *
RHEP2[5]
F[1]  *
F[1]  *
RHEP2[5]

      ρT
16                          Hence error 3 produced the "current" T.
      →                     No-argument branch clears the top suspension.

      )SI
F[1]  *
F[1]  *
RHEP2[5]
      T                     The value of T from error 2.
0 9
      )RESET                Entirely clear the state indicator.
      )SI
```

A.4 Example: Debugging With Quad Output

Here are procedures to try when an error has occurred during function execution: Check that the proper expression was entered, check near the location indicated by the error message for obvious mistakes, display relevant variables, and experiment with the line in error as described in Section A.3. These

techniques may prove inadequate on lengthy functions and functions involving loops. In that case, a simple approach is to display the results of intermediate computations with quad output. Careful selection and alteration of which results are displayed can help pinpoint the error. Section A.5 presents some additional approaches.

Consider an illustration of debugging with the use of quad output. Here is a flawed function *FAC* intended to compute factorials with a loop; *FAC* runs without stopping:

```
     ∇ Z←FAC N;K
[1]    Z←1
[2]    K←0
[3]    →4*K<N
[4]    K←K+1
[5]    Z←Z*K
[6]    →3
     ∇

       FAC 5
                    It does not stop, the user interrupts.
FAC[6]
       Z            Check some values.
1
       K
4
```

The *Z* is obviously wrong. You do not see the errors in *FAC*. Insert some quad output after line 5 to observe how *Z* changes:

```
     ∇ Z←FAC N;K
[1]    Z←1
[2]    K←0
[3]    →4*K<N
[4]    K←K+1
[5]    Z←Z*K
[6]    ⎕←Z
[7]    →3
     ∇

       FAC 5        Try again.
1
1
1
.                   Many 1s.
.
.
1                   User interrupts.

FAC[5]
```

Notice Z is not changing. Reset, then modify line 6 to display both K and Z using $\square\leftarrow K , Z$:

```
      FAC 5          Try again.
1 1
2 1
3 1
4 1
5 1
1 1
2 1
3 1
  .
  .
  .                  User interrupts.
FAC[6]
```

How can Z remain constant when K is changing? According to line 5, the value of Z should be the old Z times K. But the multiplication is not occurring. Multiplication is signified with × not the star. Correcting line 5 and line 3 and deleting the quad output fixes *FAC*.

A.5 *Setting Traces and Stops*

Consider the function *MID*, which is to be used to approximate definite integrals using the midpoint rule. Notice that *MID* is similar to *RHEP2*, except line 4 has been adjusted. Line 4 should select the midpoints of the approximating subintervals. This has not been done correctly, however, for the following illustration. *MID* is tested by applying it to a known integral, in this case to an integral giving ln(2):

```
     ∇ Z←N MID INT;A;B;DX;X;Y
[1]    A←INT[1]
[2]    B←INT[2]
[3]    DX←(B-A)÷N
[4]    X←A+¯0.5+DX×⍳N
[5]    Y←F X
[6]    Z←DX×+/Y
     ∇

     ∇ Z←F T
[1]    Z←1÷T
     ∇

      100 MID 1 2      Something is wrong.
1.09197525
      ⍟2               ln(2).
0.6931471806
      1000 MID 1 2
1.097945918
      5 MID 1 2        Small case is wrong, too.
0.9769341769
```

The next step is to introduce a *trace* vector. This results in the display of the last quantity computed on each line preceded by the function name and line number. The trace is set by assigning to a "special name" the line numbers to be traced. The special name is the function name with *T*Δ as a prefix. For example, *T*Δ*MID*←3 4 5 would set a trace on lines 3, 4, and 5 of the function *MID*; whenever those lines are executed the last result is displayed:

```
      TΔMID←⍳6          Trace all the lines of MID.
      5 MID 1 2         Run the small case again.
MID[1] 1
MID[2] 2
MID[3] 0.2
MID[4] 0.7 0.9 1.1 1.3 1.5
MID[5]   1.428571429   1.111111111   0.9090909091   0.7692307692
0.6666666667
MID[6] 0.9769341769
0.9769341769
```

Notice that the values of *X* on line 4 are wrong. This is not a selection of points from the interval [1,2]. Line 4 of *MID* is corrected and the small case is run again. The trace is removed with *T*Δ*MID*←⍳0:

```
    ∇ MID[4]                Line 4 is corrected.
[4]   X←A+DX×¯0.5+⍳N ∇

      5 MID 1 2             Notice line 4 gives the correct X.
MID[1] 1
MID[2] 2
MID[3] 0.2
MID[4] 1.1 1.3 1.5 1.7 1.9
MID[5]  0.9090909091   0.7692307692   0.6666666667   0.5882352941
0.5263157895
MID[6] 0.6919078857
0.6919078857

      TΔMID←⍳0              The trace is removed.
      100 MID 1 2           It works.
0.6931440556
      ⍟2
0.6931471806
```

The function is now working correctly.

Consider an alternate method for discovering the error. Perhaps the function *MID* is an edited version of *RHEP2* so that the offending line is very likely the edited line 4. The system can be instructed to stop before any given line numbers. The *stop* is set by assigning the lines to be stopped to the special name given by the function name with *S*Δ as a prefix. Execution will be suspended immediately before execution of the indicated lines. For example, *S*Δ*MID*←5 will request a halt before

line 5 is executed. The faulty *MID* is below; *F* is the reciprocal function as before:

```
     ∇ Z←N MID INT;A;B;DX;X;Y
[1]    A←INT[1]
[2]    B←INT[2]
[3]    DX←(B-A)÷N
[4]    X←A+¯0.5+DX×⍳N
[5]    Y←F X
[6]    Z←DX×+/Y
     ∇
```

```
      S▵MID←5                    The stop is requested.
      5 MID 1 2
MID[5]  *
      X                          Check on the value of X.
0.7 0.9 1.1 1.3 1.5

      X←A+0.5+DX×⍳N              Try another expression for X.
      X
1.7 1.9 2.1 2.3 2.5              This is wrong, too.

      X←A+DX×¯0.5+⍳N             Another try for X.
      X                          Of course!
1.1 1.3 1.5 1.7 1.9
      →5                         Finish running MID.
0.6919078857

      ∇MID[4]                    Correct line 4.
[4]   X←A+DX×¯0.5+⍳N ∇

      100 MID 1 2                Try more subintervals.
MID[5]  *
      →5
0.6931440556
      S▵MID←⍳0                   Remove the stop.
```

Appendix B
Workspace Environment

Some of the features of the APL workspace environment were introduced in Chapters 1 and 3. This appendix reviews and completes that discussion and gives some examples of managing workspaces and their contents. It ends with system commands, system variables, and system functions. Most APL systems provide additional facilities for interfacing with file systems, the operating system, and graphics displays; those features vary considerably from system to system; see your system documentation.

When an APL session begins, some computer memory is allocated for the active workspace. This provides an environment that allows the user to define variables and functions. Those functions may be executed and the variables used. These can be readily saved. The default active workspace is a "clear" workspace; that is, one that contains no user-defined functions or variables but does contain system variables with default values.

B.1 Workspace Contents: Saving and Loading

Consider an example session where some of the work from Chapter 3 had been entered into the computer. The functions and variables defined could be displayed via the system commands) *FNS* and) *VARS*. Since several functions and variables have been entered and the functions have been debugged, it is desirable to save these. These are saved with the workspace name *CHAP*3; then the workspace is cleared and *CHAP*3 is retrieved:

```
      )FNS
AVG     DF      DOT     FAC     ITERNEWT        ITERNEWT2
POLYVAL RAND    RANREAL RHEP    RHEP2
      )VARS
A       B       C       DX      N       S       X       Y
      ⎕PP←15
      )SAVE CHAP3       Saves active workspace as CHAP3.
CHAP3 SAVED 1989-02-14 02:44:54
      )CLEAR
      )FNS              No user-defined functions or
      )VARS             variables in a clear workspace.
      ⎕PP
10
```

```
      )LOAD CHAP3      Replace active workspace by CHAP3.
SAVED 1989-02-14 02:44:54
      )FNS
AVG     DF      DOT     FAC     ITERNEWT        ITERNEWT2
POLYVAL RAND    RANREAL RHEP    RHEP2
      )VARS
A       B       C       DX      N       S       X       Y
      ⎕PP
15
```

Notice that the save command caused a short message to appear. The command)*CLEAR* removed all the user-defined variables and functions and restored all system variables to their default values. The load command cleared the active workspace and loaded the functions, variables, and system variables stored in *CHAP*3. After the load command, the active workspace name is *CHAP*3. The active workspace name can be determined with the *W*ork*S*pace *ID*entification command)*WSID*.

Now suppose only the functions related to calculus computations are desired. The functions and variables from *CHAP*3 that are not needed can be erased and a new workspace, *CALC*, stored:

```
      )WSID
IS CHAP3
      )ERASE AVG DOT POLYVAL FAC RAND RANREAL X Y S A B
      )FNS
DF      F       ITERNEWT        ITERNEWT2       RHEP    RHEP2
      )VARS
C       DX      N
      )SAVE CALC
CALC SAVED 1989-02-14 02:45:59
      )WSID
IS CALC
```

Notice that the use of)*SAVE CALC* changed the name of the active workspace to *CALC*. This workspace should contain the functions *AVG* and *DOT*. Adding those functions to the active workspace can be accomplished with)*COPY CHAP*3 *DOT AVG*, where the copy command copies the two indicated functions from the *CHAP*3 workspace. Variables or system variables can also be copied:

```
      )COPY CHAP3 DOT AVG
CHAP3 SAVED 1989-02-14 02:44:54
      )FNS
AVG     DF      DOT     F       ITERNEWT        ITERNEWT2
RHEP    RHEP2
      )SAVE
CALC SAVED 1989-02-14 02:49:27
```

This last command stored the latest version of *CALC*; the previously saved version of *CALC* is overwritten.

B.2 Workspace Management

The workspaces stored on the default disk drive can be determined using the) *LIB* command. A *library* can be thought of as a collection of workspaces (and other files on some systems). If the two workspaces discussed in Section B.1 have been saved, the) *LIB* command lists them. Then a workspace is duplicated and an attempt is made to save the workspace onto an already existing workspace:

```
      )LIB
CALC      CHAP3
      )WSID
IS CALC
      )SAVE TEMP
TEMP SAVED 1989-02-14 03:01:16
      )LIB
CALC      CHAP3      TEMP
      )WSID
IS TEMP
      )SAVE CALC
NOT SAVED. THIS WS IS TEMP
```

Notice that the active workspace could not be saved with a name of another existing workspace. This is a safety feature against accidentally overwriting workspaces. On the other hand, if you want to save this workspace with the name *CALC*, you can do so by using the workspace identification command to change the name of the active workspace explicitly. Notice that a workspace can be deleted with the) *DROP* command and that the current time is noted:

```
      )WSID CALC
WAS TEMP
      )SAVE          Save workspace with previously assigned name.
CALC SAVED 1989-2-14 03:02:11
      )LIB
CALC      CHAP3      TEMP
      )DROP TEMP
1989-02-14 03:02:36
      )LIB
CALC      CHAP3      Workspace TEMP no longer exists.
```

The final topic in this section is an exercise in transferring workspaces between disk drives. Transfers between libraries on a time-sharing system are analogous. Suppose a friend has provided several public domain workspaces on a floppy and suggested that you put copies of the *LINALG* and *NUMBERTH* workspaces onto your hard disk. Suppose the floppy disk is drive 1 and the hard disk is drive 3. Here is a sample of how the workspaces can be transferred using APL commands:

```
      )LIB 1
LINALG   NUMBERTH TEST      UTILITY
      )LOAD 1 LINALG
```

```
1 LINALG SAVED 1986-10-17 15:10:11
      )SAVE 3 LINALG
3 LINALG SAVED 1989-02-14 03:04:19
      )LOAD 1 NUMBERTH
1 NUMBERTH SAVED 1985-05-09 09:10:37
      )SAVE 3 NUMBERTH
3 NUMBERTH SAVED 1989-02-14 03:04:21
      )LIB 3
CALC      CHAP3      LINALG     NUMBERTH
```

That completes the transfer of the two desired workspaces.

B.3 System Commands

)CLEAR	Replaces the active workspace by a clear workspace, which contains no user-defined variables or functions and has system variables set to the default values.
)CONTINUE	Saves the active workspace with the name CONTINUE and exits APL. Some systems provide an automatic save to the CONTINUE workspace when a connection is lost. The workspace is reloaded via)LOAD CONTINUE. Since it is easy to overwrite important work when next using the CONTINUE workspace, a save command)SAVE, with a good choice of name, followed by)OFF is recommended instead of)CONTINUE.
)COPY ABC F G X	Copies the objects (functions, variables, or groups) F, G, and X from the workspace ABC into the active workspace.
)COPY 1 ABC F G	Copies the objects F and G from the workspace ABC from disk drive 1 or account number 1.
)DROP ABC	Deletes the workspace ABC from the default disk drive or account number.
)DROP 2 ABC	Deletes the workspace ABC from disk drive 2 or account number 2.
)ERASE X TT A	Deletes the objects X, TT, and A from the active workspace.
)GROUP AB X F Y	Associates the objects X, F, and Y as members of a group with the name AB. This group of functions or variables may be copied or erased by referring to the group name AB. Not all modern APL systems support "group" commands.
)GRP AB	Lists the members of the group AB.
)GRPS	Displays the names of the groups in the active workspace.
)HELP	Invokes the help utility on systems with this facility.
)LIB	Lists the names of the workspaces on the default disk drive or account number.

) *LIB 2*	Lists the workspaces on disk drive 2 or account number 2.
) *LOAD ABC*	Replaces the active workspace with the workspace *ABC* from the default drive or account.
) *LOAD 1 ABC*	Loads the workspace *ABC* from disk drive 1 or account number 1.
) *OFF*	Exits APL. The active workspace is lost; it should be saved, if so desired, before exiting APL.
) *PCOPY ABC A F*	Protects during copying. Copies from the workspace *ABC* the objects *A* and *F*; objects with the same name in the active workspace are not overwritten. The disk drive number of the workspace may also be included.
) *RESET*	Resets the state indicator. The suspended functions are all restored to a normal state. Workspace variables are all available for use and none of the automatic debugging facilities, is invoked. On some systems this is) *SIC*, *S*tate *I*ndicator *C*lear.
) *SAVE*	Saves the active workspace with the current name. The defined variables and functions, system variables, and suspended functions in the active workspace are saved.
) *SAVE ABC*	Saves the active workspace with the name *ABC* and assigns the name *ABC* to the active workspace.
) *SI*	*S*tate *I*ndicator lists the suspended and pendant functions. See also) *RESET* and) *SIV* and Appendix A.3.
) *SIV* or) *SINL*	Lists the suspended and pendant functions and the variables local (names local) to each.
) *SYMBOLS*	Results in the number of symbols for which room exists in the symbol table. This is automatically increased as needed in modern systems.
) *WSID*	Results in the name of the active workspace.
) *WSID ABC*	Gives the active workspace the name *ABC*. A disk drive or account number may be used before the workspace name.
) *XLOAD ABC*	Loads the workspace *ABC* while suppressing the automatic execution of □*LX* (see Section B.4).

B.4 System Variables

□*CT*	*C*omparison *T*olerance is used for comparison of floating point numbers. It gives the relative precision required. The default value of □*CT* is about $2^{-46} \approx 1.4 \times 10^{-14}$. See Section 5.12.
□*IO*	*I*ndex *O*rigin determines whether indexing of arrays and axes begins with index 0 or 1. The default value of □*IO* is 1. See Section 9.7.

⎕LX — *L*atent e*X*pression is a character vector, assignable by the user, which is automatically executed each time the workspace is loaded. In a clear workspace it is the empty vector.

⎕PP — *P*rint *p*recision determines the number of base 10 significant figures that are displayed. Typically, the meaningful range is 1 to 17. The default value ⎕*PP* is 10. See Section 1.16.

⎕PW — *P*rint *W*idth specifies the width of the session log display. This is the number of characters that can be displayed before wrapping to the next line is required. On many systems, the default width is 80.

⎕RL — *R*andom *L*ink is the "seed" number used for random number generation. In a clear workspace this is 16087. See Section 6.1.

B.5 System Reports and System Functions

⎕AI — *A*ccount *I*nformation contains several numbers in a vector. Typically, these are account number, CPU time this session, time, number of characters of input and output this session, and so on. See your system documentation.

⎕AV — *A*tomic *V*ector is a vector of all the characters. On typical systems this gives all 256 possible bytes that can be manipulated as character data.

⎕CR 'AVG' — *C*anonical *R*epresentation produces a character matrix that gives the definition of the function named in the character vector that is given as the right argument. Here the result would give the definition of the function *AVG*. The canonical representation includes the header but not the line numbers.

⎕DL 5 — *D*e*L*ay causes a wait of at least the indicated number of seconds, here 5. The result is the actual delay.

⎕ER — *E*vent *R*eport gives a report of the last occurrence of an error.

⎕EX 'T AB' — *EX*punges (erases) the most local use of the variables in its right argument. Here the variables *T* and *AB* would be erased. This could be used within a function to free the memory used by a temporary variable once the variable is no longer needed.

⎕FI '25.1 3T 6' — *F*ield *I*nput is used to interpret a character vector as numeric data. Non-meaningful fields result in 0. In the given example the result would be 25.1 0 6 since 3*T* is not a meaningful representation of a number. See also ⎕*VI*.

⎕FMT — For*MaT* is used for formatting the display of data using several formatting phrases. For example, if *A*←2 3⍴(⍳6)÷3 then

```
      'F10.2'⎕FMT A
0.33      0.67      1.00
1.33      1.67      2.00
```

The formatting phrases include *Fm . n*, which gives fixed point formatting in fields of width *m* and with *n* figures to the right of the decimal point. Exponential formatting is obtained with *Em . n* and integer formatting is obtained with *Im*. Formatting by example is accomplished with *G* formats. The formats are given as a character string in the left argument and may be preceded by repetition factors and separated by commas. Thus with the same *A* as above,

```
      '2E10.2,I2'⎕FMT A
3.33E¯1   6.67E¯1  1
1.33E0    1.67E0   2
```

⎕FX CHAVG — *FiX* is used to define a function whose canonical representation is given as the right argument. For example, if *CHAVG*←⎕*CR* '*AVG*', then ⎕*FX CHAVG* can be used to define *AVG*. This is especially valuable for defining functions from within function control.

⎕LC — *L*ine *C*ounter gives a vector of line numbers; it begins with the currently executing function line, which is followed by the line number(s) of functions to which that function is pending. This is the empty vector in immediate execution mode.

⎕LOAD 'ABC' — *LOAD*s the workspace *ABC*. A disk drive number or an account number may be inserted before the workspace name. This can be used from function control, and the workspace name can be computed during function execution.

⎕NC 'F1 A' — *N*ame *C*lass gives the class of the names listed in the right argument. The right argument may be a character vector as in the example or a character matrix with one name per row. The result is a numeric vector with entries corresponding to the names as follows:

0 not in use
1 used as a label
2 used as a variable
3 used as a function
4 group or other use

In the given example, the result would be the vector 3 2 if *F1* is a function and *A* is a variable.

⎕NL 2 — *N*ame *L*ist results in a matrix of the names of the objects in the indicated class in the active workspace. See ⎕*NC* for the meaning of the indicated class. The given example would result in a matrix with rows that give the names of the variables in the active workspace.

⎕QLOAD 'ABC' — *Q*uiet *LOAD*s the workspace *ABC*; that is, loads *ABC* without displaying the date saved message.

⎕TS — *T*ime *S*tamp gives the current time as a numeric vector with entries indicating the year, month, day, hour, minute, second, and millisecond.

`⎕UL` *U*ser *L*oad gives the current number of users.

`⎕VI '25.1 3T 6'` *V*alidate *I*nput takes a character argument as its right argument and results in a Boolean vector indicating whether or not the fields in the character argument represent meaningful numeric entries. Here the result would be `1 0 1` since `3T` is not a meaningful representation of a number. See also `⎕FI`, which results in a vector of the numerical values represented by the character string.

`⎕WA` *W*ork *A*rea results in the number of bytes of memory currently available in the active workspace.

Appendix C
Keyboards

1. The characters shown in Figure C.1 are formed directly or with use of the shift key as indicated at the top left of the figure.
2. Overstrike characters are formed using the backspace key. For example, ⍜ is formed by ○, ⟨backspace⟩, \.

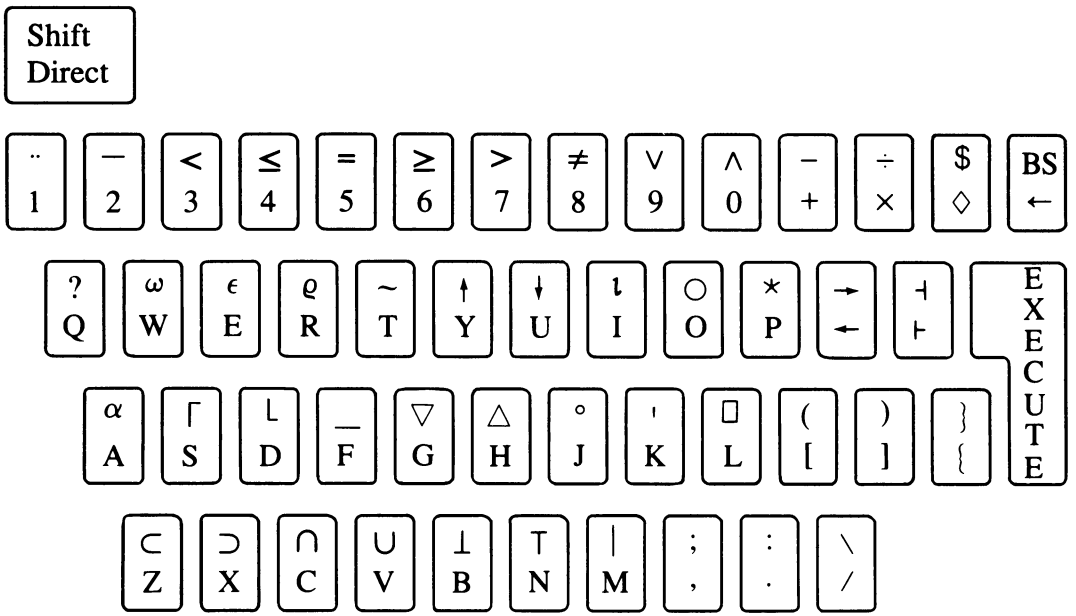


Figure C.1 An APL keyboard.

1. The characters shown in Figure C.2 are formed directly and with use of the shift and alternate keys as indicated at the top left of the figure.
2. Overstrike characters are formed with the alternate backspace key. For example, ⍜ is formed by ○, ⟨alternate backspace⟩, \.
3. The primary alphabet is lowercase here so that system commands, system variables, and exponential notation are given as)off, ⎕pp, and 1.23e45 rather than as)OFF, ⎕PP, and 1.23E45.

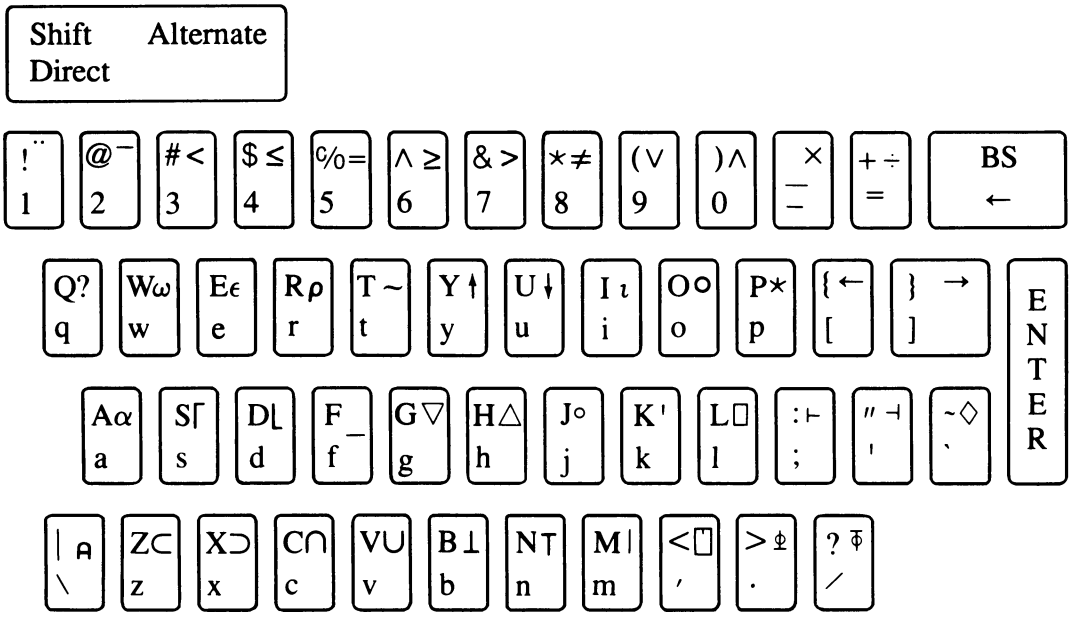


Figure C.2 Union keyboard (used by I. P. Sharp Associates).

1. The characters shown in Figure C.3 are formed directly and with use of the shift and alternate keys as indicated at the top left of the figure.
2. Overstrike characters are not needed.
3. System commands and system variables may use either uppercase or lowercase as in)OFF,)off, ⎕PP or ⎕pp, but exponential notation requires a capital E as in 1.23E45.

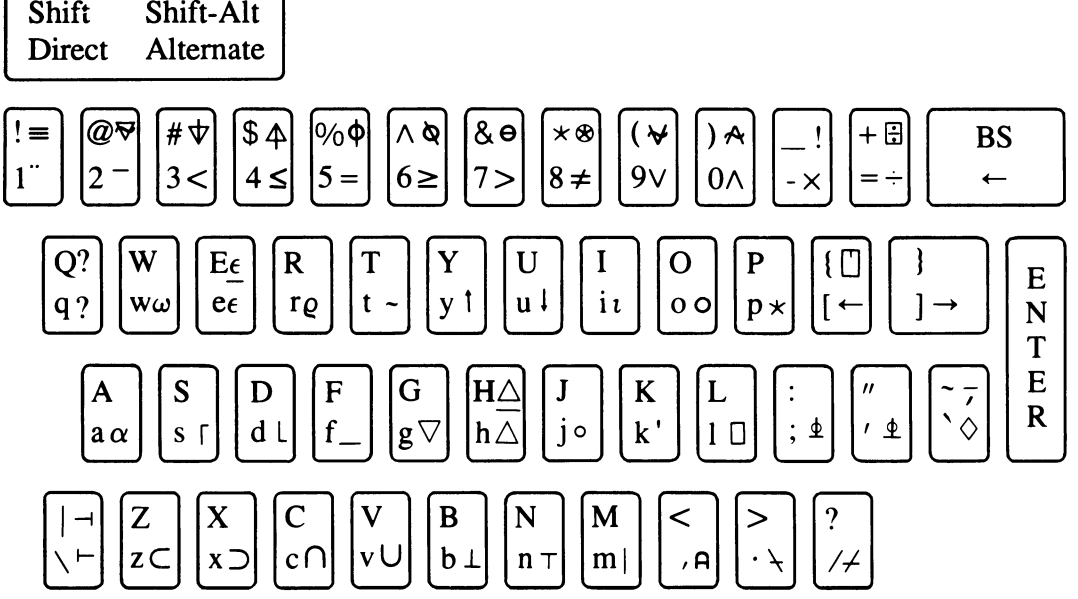


Figure C.3 Union keyboard (used by STSC, Inc.).

Appendix D
Answers to Selected Exercises

Chapter 1

1. (c) 2 (d) 8
(g) 1 (h) 0
(k) 0 (l) 0
(o) 0.1 (p) 2.718281828
(s) 3 (u) ¯1

2. (c) ¯6 (d) 2
(g) 6 (h) ¯1
(k) 1 (l) 3

4. (a) 3 3 (b) 6 6
(c) ¯3 ¯2 ¯1 0 1 2 3 (d) 3 2 1 0 ¯1 ¯2 ¯3
(i) 1 (j) ¯1 ¯2 ¯4
(k) 3 (*L*) 8

6.

```
      B×U      +/B×U yields the sum of the odd entries in B.
1 0 3 0 5 0 7 0 9 0 11 0 13
      B*U      ×/B*U yields the product of the odd entries in B.
1 1 3 1 5 1 7 1 9 1 11 1 13
```

7. (a) domain error (b) domain error (e) syntax error
(f) length error (i) syntax error (j) domain error

9. (a) .8+.2×⍳16

10. (a) +/÷⍳100

11. (a) +/*V*×*V* or +/*V**2 (b) +/|*V* (d) +/0=*V*

13. Moment about *x* axis: +/*M*×*X* yields 2.55
Moment about *y* axis: +/*M*×*Y* yields 8.405.
Center of mass, *x* coordinate: (+/*M*×*X*)÷+/*M* yields 0.5257731959.
Center of mass, *y* coordinate: (+/*M*×*Y*)÷+/*M* yields 1.732989691.

Chapter 2

2. (a)
```
2 4 4
6 6 8
```
(b)
```
1 1 3
  2 5 3
```

```
(c)  1  4  3      (d)  4 5 4
     16 25 36           5 4 5
(e)  1 1 0        (f)   2  4  8
     0 0 0             16 32 64
```

3. (a) `3 4⍴⍳4` (b) `3 4⍴⍳3` (c) `3 2⍴1` (d) `3 1⍴⍳3`

5. (a), (d), (f) give 4 5; (b), (c), (d), (e) give 3 3 3.

6. (b) `1 2 3∘.+1 2 3 4 5`

8. (b) `2 ¯3 ¯4` (d) `2 3 2 3 2 3` (f) `6 3 3`

10. (a) and (c)
```
1 1 1 2 3 4
1 1 1 2 3 4
```
(b) and (d)
```
1 1 1
1 1 1
2 3 4
2 3 4
```

11. (a) and (c)
```
1 1 1 2 2
1 1 1 2 2
```
(b) and (d) give a length error

13. (a)
```
 4  2  5
 0  1  2
¯4  3  1
```
(b)
```
¯4  3  1
 4  2  5
 0  1  2
```

14. (a)
```
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
```
(b) `2 3 4` (c) `3` (d) `2` (e) `5` (f) `13` (g) `1 2 3 4` (h) `1 5 9` (i) `1 13`

(j)
```
1  2  3  4
5  6  7  8
9 10 11 12
```
(k)
```
 1  2  3  4
13 14 15 16
```
(l)
```
 1  5  9
13 17 21
```

(m) and (q)
```
14 16 18 20
22 24 26 28
30 32 34 36
```
(n)
```
15 18 21 24
51 54 57 60
```
(o) and (p)
```
10 26 42
58 74 90
```

(r) and (v)
```
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

 3  3  3  3
 3  3  3  3
 3  3  3  3
```
(s)
```
 1  2  3  4
 5  6  7  8
 9 10 11 12
 3  3  3  3

13 14 15 16
17 18 19 20
21 22 23 24
 3  3  3  3
```

(t) and (u)
```
 1  2  3  4  3
 5  6  7  8  3
 9 10 11 12  3

13 14 15 16  3
17 18 19 20  3
21 22 23 24  3
```

16. (c)
```
3 4
7 8
```
(d)
```
7 8
3 4
```

17. (a)
```
1  0  0 ¯1
0  1  0  1
0  0  1  4
```

19. **(a)** 7 4 **(b)** 3 4
(c) 2 4 **(d)** 4 8
(e) 4 **(f)** 4 2 3

20. ρ*V*[*I*] is ρ *I*

21. **(a)** 3 4 **(b)** 3 (one element vector)
(c) empty vector **(d)** 0 (one element vector)
(e) (empty vector) **(f)** (empty vector)
(g) (empty vector)

22. **(a)** 15 **(b)** 10 **(c)** 5 **(d)** 0 **(e)** 0

23. **(a)** 2 **(b)** 1 **(c)** empty; shape 3 0, dimension 2 **(e)** 8
2 2 **(d)** empty; shape 4 0, dimension 2 8
2 5 8
(f) empty; shape 0 5, dimension 2 **(g)** empty; shape 0 4, dimension 2
(h) empty; shape 0 1, dimension 2 **(i)** 3 (one element matrix)
(j) empty; shape 0 0, dimension 2

Chapter 3

1.
```
∇ Z←SD V
[1]   Z←((+/(V-(+/V)÷ρV)*2)÷¯1+ρV)*.5
∇
```

3.
```
∇ Z←E DIST P0
[1]   Z←(|+/E×P0,1)÷(+/E[⍳3]*2)*0.5
∇
```

5. **(a)** 4.84 **(b)** 4.0804
(c) 4.008004 **(d)** ¯3.9204
(e) ¯3.992004 **(f)** 4.008004

7.
```
∇ Z←N RANINTERVAL INT;A;B
[1]   A←INT[1]
[2]   B←INT[2]
[3]   Z←A+(1E¯17×B-A)×?Nρ1E17
∇
```

9. **(a)** *PARTITION* creates a vector of endpoints of the subintervals of its right argument. *LHEP* approximates a definite integral using rectangles with heights determined at the left-hand endpoints.

(b)
```
∇ Z←N RHEP AB
[1]   Z←((AB[2]-AB[1])÷N)×+/(0,Nρ1)×F N PARTITION AB
∇
```

(c)
```
∇ Z←N TRAP AB
[1]   Z←((AB[2]-AB[1])÷2×N)×+/(1,((N-1)ρ2),1)×F N PARTITION AB
∇
```

(d)
```
∇ Z←N SIMP AB
[1]   Z←((AB[2]-AB[1])÷N×6)×+/(1,((N-1)ρ4 2),1)×F N PARTITION AB
∇
```

12. $X_0 = 2, X_{10} = {}^{-}1.105719329\ F(X_{10}) = {}^{-}9.853229344E^{-}16$
$X_0 = 8, X_{10} = 4.949418444\ \ F(X_{10}) = 3.552713679E^{-}15$

13.
```
∇ Z←POLYDER P;E
[1]   Z←P[1+E]×E←⍳¯1+ρP
∇
```

15. Here *D* gives the number of terms in the resulting polynomial, and *C* give the coefficients that are multiplied times each term.

```
      ∇ Z←N POLYHDER2 P;C;D
[1]     D←(⍴P)-N
[2]     C←×/(¯1+⍳D)∘.+⍳N
[3]     Z←C×P[N+⍳D]
      ∇
```

16. **(b)** *ITER2* *N* gives the sum of the first *N* positive integers (0 if *N* is 0)
(d) *ITER4* *N* gives a list of the first *N* "triangular" numbers (the *K*th triangular number is the sum of the first *K* positive integers; that is, +/⍳*K*). (Empty when *N* is 0.)
(f) *ITER6* *N* gives 0 if *N* is not 0; it is 1 if *N* is 0.

18. $X_0 = 2, X_1 = 8, X_{15} = {}^-1.105719329, F(X_{15}) = {}^-9.853229344E^-16$

20.
```
      ∇ Z←FIB N;K;OLDZ;NEWZ
[1]     Z←0 1 ◊ K←1
[2]     →3×K<N
[3]     Z←Z,+/Z[K+0 1] ◊ K←K+1
[4]     →2
      ∇
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
      6765 10946 17711 28657 46368 75025 121393 196418 317811
      514229 832040 1346269 2178309 3524578 5702887 9227465
      14930352 24157817 39088169 63245986 102334155 165580141
      267914296 433494437 701408733 1134903170 1836311903
      2971215073 4807526976 7778742049 12586269025
```

Chapter 4

1.
```
4 ¯1 2 ¯1 POLYVAL 30 PARTITION ¯1 2
```

3.
```
      ∇ Z←VANDERMONDE X
[1]     Z←X∘.*¯1+⍳⍴X
      ∇
```

5.
```
(⍳10)∘.=⍳10
```

8. **(a)** $-216 + 265.6t$ **(d)** $-4 - 2t + 2t^3 + 4t^4$

9. **(a)** *U*+.=*V* yields the number of positions in which the vectors have the same component.
(d) *A*+.×*V* yields the sum of all the numbers from 1 to 100 that are either multiples of 5 or 2 more than multiples of 5. *A*×.**V* yields the product of the same set of numbers.

10. **(b)**
```
¯1 ¯2 ¯3 ¯4
```
(c)
```
¯1
¯2
¯3
¯4
```
(i)
```
1  5 ¯1
2  6 ¯2
3  7 ¯3
4  8 ¯4
```

13. **(a)**
```
0.479623078 0.7871183218 0.387822705
```
(b)
```
11.79886983
```
(c)
```
¯4.208802418E¯9 5.132452241E¯9 ¯5.211679976E¯9
```

14. **(a)**
```
V←0.0008488338378 ¯0.5206534552 0.8537676845
```
(b) 0.5168397689 using *V*+.×*A*1+.×*V*
(c)
```
1.333715873 ¯0.4785159911 ¯0.2931395881
```
(d)
```
¯0.7224497355 ¯0.06356822402 0.6884950694
¯1.442466986
 0.01269801795 ¯0.01360978565 0.01206766784
```

15. **(a)**
```
0.5481205682 ¯0.615289491 0.5665533382
```
(b)
```
1.644333726
```
(c)
```
0.00000338887139 3.912404372E¯7 ¯2.853718924E¯6
```
(d)
```
¯0.7265445758 ¯0.05915772283 0.6845679976
¯1.443203669
 1.499485549E¯7 ¯1.623066018E¯7 1.451172427E¯7
```

16. **(a)**
```
-0.5459938902  0.6171913235 ¯0.5665382089
```
(b)
```
1.644319472
```
(c)
```
0.008413063693 0.01853637067 0.01208569107
```

18.
```
      ∇ Z←GRAMSCH A;K;N
[1]     K←0 ◊ N←(⍴A)[2] ◊ Z←A[;⍳0]
[2]     →3×K<N
[3]     K←K+1
[4]     Z←Z,UNIT A[;K]-A[;⍳K-1]+.×A[;K]⌹A[;⍳K-1]
[5]     →2
      ∇
```

19.
```
      ∇ Z←N QR A;Q;K
[1]     K←0 ◊ Z←A
[2]     →3×K<N
[3]     Z←(⍉Q)+.×Z+.×Q←GRAMSCH Z
[4]     K←K+1
[5]     →2
      ∇
```

```
      5 QR A1
11.799      ¯0.68827     ¯1.5867
 0.00097477  1.1477      ¯1.1753
 0.00027704 ¯1.0948      ¯0.94664
      10 QR A1
 1.1799E1     3.6112E¯1    1.6921E0
 4.3038E¯8    1.5142E0    ¯5.8159E¯1
 8.1725E¯9   ¯6.6192E¯1    ¯1.3130E0
      15 QR A1
 1.1799E1    ¯2.1565E¯1    ¯1.7167E0
 2.1908E¯12   1.5945E0     ¯4.3133E¯1
 2.3392E¯13  ¯3.5100E¯1    ¯1.3934E0
      20 QR A1
 1.1799E1     8.7254E¯2    1.7280E0
¯5.5008E¯16   1.6379E0     ¯1.0594E¯1
¯9.5443E¯15  ¯1.8627E¯1    ¯1.4368E0
```

Chapter 5

2. **(a)** `+/D<|M-V` **(b)** `+/D>|M-V`

4.
```
   +/((J<V)∧V<K)∨(M<V)∧V<N
```
or
```
+/∨≠((J,M)∘.<V)∧(K,N)∘.>V
```

6. Counts the entries of V that lie in at least one of the intervals.

8. **(a)** Boolean vector with 1s marking the positions of 2s in V.
(b) Boolean vector with 1s marking the positions of 2s, 5s, and 9s in V.
(c) Count of the number of 2s, 5s, and 9s in V.

10.
```
(+/S×S≥10)÷+/S≥10
```

12. **(d)** Is at least one entry of *U* less than or equal to the entry in the corresponding position of *V*?
(e) Is *U* different from *V* in at least one position?
(g) Is every entry of *V* positive?

14. To three decimal places: 2.854

16.
```
.1×⌊.5+1000×S÷650
```

18.
```
      ∇ Z←WARSHALL R;K;N
[1]     N←(⍴R)[1] ⋄ K←1
[2]     Z←R∨R[;K]∘.∧R[K;]
[3]     →4×∨/∨/Z≠R
[4]     R←Z
[5]     →2×N≥K←K+1
      ∇
```

Chapter 6

2.
```
(5>+/R)∨6∨.=R←?6 6 6
```

4.
```
      ∇ Z←AVGMATCH N;I
[1]     Z←⍳0 ⋄ I←0
[2]     →3+3×N=I
[3]     Z←Z,+/(20?20)=⍳20
[4]     I←I+1
[5]     →2
[6]     Z←(+/Z)÷N
      ∇
```

```
⎕RL←16807
AVGMATCH 100
0.9
```

6. Changes are in the header and line 1 and 3.

```
      ∇ Z←V POLYA U;B;D;I;K;N;R;T;W
[1]     R←V[1] ⋄ W←V[2] ⋄ K←V[3] ⋄ D←U[1] ⋄ N←U[2]
[2]     T←R+W ⋄ I←0 ⋄ Z←⍳0
[3]     →4×I<D
[4]     B←R≥?N⍴T ⋄ Z←Z,(+/B)÷N
[5]     I←I+1 ⋄ R←R+K×B ⋄ T←T+K
[6]     →3
      ∇
```

8. **(a)**
```
¯3.544907702 1.772453851 0.8862269255 4.260820476
 7.188082729
```
(b) domain error **(c)** domain error

12.
```
⎕RL←16807
X←RANREAL 1000 ⋄ Y←(2*.5)×RANREAL 1000 ⋄ .001×+/Y<4○X*2
0.779
```

Chapter 7

1. **(a)** 1 ¯4 3 ¯2 **(b)** 1 0 5 0
(c) 5 **(d)** 6 7 5
(e) 3 9 6 7 **(f)** 0 3 ¯1 ¯3
(g) 9 **(h)** 2
(i) 5

3.
```
((+/V)-(⌈/V)+⌊/V)÷¯2+⍴V   7.666666667
```

4. **(a)** `2 4 1 3` **(b)** `¯2 0 1 3`

5. **(a)**
```
AEIOUA
EIOUAE
IOUAEI
```
(d)
```
ADE
BCB
```

7. **(a)** `'*∘'[1+(⍳4)∘.<⍳4]` **(b)** `'*∘'[1+(⍳4)∘.≠⍳4]`
(c) `'*',(3 3⍴'∘'),[1]'*'` **(d)** `(⍳5)/'+-=<|'`

10. Need ρU to be the same as ρV. The result is a vector with $+/U$ entries. That does not depend on V.

12.
```
     ▽ Z←MODE W;A;F;M
[1]    A←M+¯1+⍳(⌈/W)-M←⌊/W
[2]    F←+/A∘.=W
[3]    Z←(F=⌈/F)/A
     ▽
```

14.
(a) `1 3 6 10 15 21 28 36 45 55`
(b) `1 2 6 24 120 720 5040 40320 362880 3628800`
(c) `1 ¯1 2 ¯2 3 ¯3 4 ¯4 5 ¯5`
(d) `2 2 2 2 5 5 5 7 7 7`
(e) `2 0 ¯2 ¯2 ¯2 ¯2 ¯2 ¯2 ¯2 ¯3`

17.
```
     ▽ Z←PLOTXYC XY;R;S;X;Y
[1]    R←⌈/X←1+XY[1;]-⌊/XY[1;]
[2]    S←⌈/Y←1+XY[2;]-⌊/XY[2;]
[3]    Z←(S×R)⍴' '
[4]    Z[X+R×S-Y]←'*∘+o▽×#. '[XY[3;]]
[5]    Z←(S,R)⍴Z
     ▽
```

20.
```
     ▽ Z←N PLOT2FCT AB;Y1;Y2
[1]    Y1←⌊0.5+FCT1 X←(N-1) PARTITION AB
[2]    Y2←⌊0.5+FCT2 X
[3]    Z←PLOTXYC ((3,N)⍴(⍳N),Y1,N/1),(3,N)⍴(⍳N),Y2,N/2
     ▽
```

22.
```
     ▽ Z←SC SCHIST W;M;A;F
[1]    A←M+¯1+⍳1+(⌈/W)-M←⌊/W
[2]    F←⌊0.5+(÷SC)×+/A∘.=W
[3]    Z←' *'[1+(M+1-⍳M←⌈/F)∘.≤F]
     ▽
```

23. Using `⎕RL←16807`, average: `1.52`,
standard deviation: `0.9997979594`

```
2 SCHIST X21
 *
 **
 **
 **
 **
 **
 **
 **
****
****
****
****
****
****
****
****
*****
```

25. Using □*RL* ←16807, average: 99.30638036
standard deviation: 11.70147525

```
HIST ⌊.5+X23
                                        *
                                 **     *
                                 **     *
                    *            **  *  **  **     *
                    *  *    *    *** *  **  *****  **
                    *  *****  ******* ********* ****  **
*     *       * * **  ****** ******* ********** ********* *  *
```

28. (a)

```
     ∇ Z←SAMPQ R;K;X;Y
[1]    K←0 ◊ Z←⍳0
[2]    →3×K<R
[3]    X←0 30 SAMPUNI 1
[4]    Y←0 0.5 SAMPUNI 1
[5]    →3+3×Y<(X×30-X)÷4500
[6]    Z←Z,X ◊ K←K+1
[7]    →2
     ∇
```

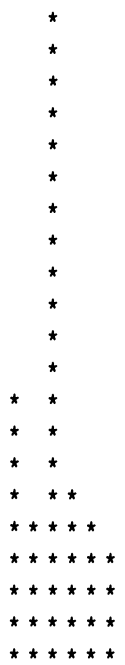
(b)

```
                *
                *
            *   *
  *  *      ****     ***     *  *
  *  *      *****    ****    *  *
*******  ************ *  *  *
```

29. **(a)**

```
∇ Z←PD SAMPD R;C;IN
[1]   C←+\PD[2;]
[2]   IN←1+(⍳⍴C)⌈.×C∘.<RANREAL R
[3]   Z←PD[1;IN]
∇
```

(b)



31.

```
∇ Z←MS SAMPNORM R
[1]   Z←MS[1]+MS[2]×(1○2×○RANREAL R)×(¯2×⍟RANREAL R)*0.5
∇
```

Chapter 8

1. **(a)** 1 1 2 3 **(b)** 5 8
(c) 2 3 5 8 **(d)** 1 1
(e) 1 1 2 3 5 8 0 0 **(f)** 0 0 1 1 2 3 5 8
(g) empty vector

2. **(a)** and **(c)** *A* must be a vector of integers with as many components as *B* has axes.
(b) ⌈/*A* **(d)** 0⌈(⍴*B*)−⌈/*A*

4. **(a)** The function *POLYADD* (Section 8.2), which was written for adding polynomials in one variable, works also for polynomials in two variables. How about three?
(b)

```
8  0  0  0  1
3  0  5  0  0
4  0  3  0  0
```

or $8 + 3x + 4x^2 + 5xy^2 + 3x^2y^2 + y^4$

6. ,⌿×/⍴*A*

7. Replace line 1 by *M*←(⍴,*A*)⌈⍴,*B*.

9. **(a)**
```
0 1 1 2 3 3 0
1 2 2 3 0 0 1
2 3 3 0 1 1 2
```
(c)
```
0 1 2 3 0
1 2 3 0 1
1 2 3 0 1
2 3 0 1 2
2 3 0 1 2
2 3 0 1 2
```
(e)
```
0 0 0 1 0 0 2 0 0 3 0 0 0
1 0 0 2 0 0 3 0 0 0 0 0 1
2 0 0 3 0 0 0 0 0 1 0 0 2
```

10. **(a)** *U* may be a scalar or a vector of nonnegative integers matching the length of the last axis of *A*.
(b) *U* must be a Boolean vector having as many 1s as the length of the last axis of *A*.
(c) *U* may be a scalar or a vector of nonnegative integers matching the length of the first axis of *A*.
(d) *U* must be a Boolean vector having as many 1s as the length of the first axis of *A*.

14.
```
     ∇ Z←U POLYMULT V
[1]    Z←+⌿(1-⍳⍴U)⌽U∘.×V,0×1↓U
     ∇
```

15. It yields the vector of coefficients of the product polynomial in decreasing power order.

17. **(a)**
```
0 1 2 3 0
2 3 0 1 1
0 1 2 2 3
```
(b)
```
0 2 0 3 1
1 3 2 0 2
2 1 3 1 0
```
(c)
```
0 1 2 3 0
1 1 2 3 0
1 2 2 3 0
```
(d)
```
1 2 3 0 0
2 3 0 1 1
3 0 1 2 2
```
(e)
```
0 3 2 1 0
1 0 3 2 1
2 1 0 3 2
```
(f)
```
2 3 0 1 2
1 2 3 0 1
0 1 2 3 0
```

19. **(a)** *R* must be an array of integers having shape ¯1↓⍴*A*; that is, the shape of *A* less its last component.
(b) ⍴*A*

20.
```
2 1 3⍉A
```

22.
```
     ∇ Z←DIAG V;N
[1]    N←⍴V
[2]    Z←(-⍳N)⌽((N,N-1)↑0),V
     ∇
```

23. **(a)** 5 12 **(b)** 2 5 6
(c) 10 6 **(d)** 5 2 6
(e) 5 12 **(f)** 5 6 2

Chapter 9

2. **(a)** Gives the same vector as ⍋*V*.
(b) Gives the same vector as ⍋⍋*V*.
(c) Gives the ranks of the elements with the highest element of *V* ranked first.

3. 1↑⍴*A*

4. *GPA*[⍋*GPA*[;2];] reorders the rows of *GPA* so the second column is in increasing order and the original order is preserved when ties occur in the second column.
GPA[⍒*GPA*[;3];] reorders the rows of *GPA* so the third column is in decreasing order and the original order is preserved when ties occur in the third column.
GPA[⍒*GPA*[;4];] reorders the rows of *GPA* so the fourth column is in decreasing order and the original order is preserved when ties occur in the fourth column.

GPA[⍒*GPA*[;4 3];] reorders the rows of *GPA* so the fourth column is in decreasing order and ties are decided by putting the third column in decreasing order. Moreover, the original order is preserved when ties occur in the fourth and third columns.
GPA,[⍪⍒*GPA*[;4 3]] gives the matrix with the rankings according to the ordering in the previous part.

8.

```
    ∇ Z←ALPHAB A;ALF
[1]   ALF←'AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz'
[2]   Z←A[⍋ALF⍳A;]
    ∇
```

9. **(a)** 0 4 8 7
(b) 8 8 0 4 8 7 8 5 3 1
(c) 0 1 0 3 4 5 1 7

11. **(a)** Need *A* to be a vector; that is, 1=⍴⍴*A*.
(b) ⍴*B*

13.
```
    ∇ Z←NUB W
[1]   Z←((W⍳W)=⍳⍴W)/W
    ∇
```

16.
```
    ∇ Z←HIST W;M;A;F
[1]   A←M+⍳1+(⌈/W)-M←⌊/W
[2]   F←+/A∘.=W
[3]   Z←' *'[(M-⍳M←⌈/F)∘.≤F]
    ∇
```

18.
```
    ∇ Z←P POLYVAL T
[1]   Z←P+.×T*(⍳⍴T)-⎕IO
    ∇
```

19.
```
      1+0+.=X∘.|X←1+⍳99
 2 2 3 2 4 2 4 3 4 2 6 2 4 4 5 2 6 2 6 4 4 2 8 3 4 4 6 2 8 2 6 4
     4 4 9 2 4 4 8 2 8 2 6 6 4 2 10 3 6 4 6 2 8 4 8 4 4 2 12 2
     4 6 7 4 8 2 6 4 8 2 12 2 4 6 6 4 8 2 10 5 4 2 12 4 4 4 8 2
     12 4 6 4 4 4 12 2 6 6 9
```

21. **(c)** The multiplication table modulo 6 has zero entries other than those in the first row and column.

24. **(a)** 2 2 2 2⊤⍳2*4 (use ⎕*IO*←0)
(b) 3 3 3⊤⍳3*3

26. +/∧⌿*X*=⍉*X*←4 *NUMBASE N* 4 9 16 25

27. **(a)** No condition
(b) (⍴*V*),⍴*B*

28. **(a)** (⍴*V*)=1↑⍴*B*
(b) 1↓⍴*B*

29. **(a)**
```
    ∇ Z←S2HMS SEC
[1]   Z←24 60 60⊤SEC
    ∇
```
```
     0  0  1 13
     0  8 23 53
    50 20 20 20
```

30.
```
    ∇ Z←HAM NMES;Q
[1]   Q←⍉2 2 2 2 2⊤13 11 8 7 4 2 1
[2]   Z←Q≠.∧2 2 2 2 2⊤16 16⊤⎕AV⍳NMES
    ∇
```

31.
```
    ∇ Z←CORHAM RMES;M
[1]   M←2 2 2⊤4 2 6 1 5 3 7
[2]   Z←RMES≠(⍳7)∘.=(2⊥M)⍳2⊥M≠.∧RMES
    ∇
```

Chapter 10

1. **(a)**

```
∇ Z←ARITHMEAN V
[1]   Z←(+/V)÷⍴V
∇
```

(b)

```
∇ Z←GEOMEAN V
[1]   Z←(×/V)*÷⍴V
∇
```

(c)

```
∇ Z←HARMEAN V
[1]   Z←(⍴V)÷+/÷V
∇
```

4.

```
∇ Z←BORDA B;S;T
[1]   Z←0
[2]   →(1<+/S=T←⌊/S←+/B)/0
[3]   Z←S⍳T
∇
```

BORDA B yields 3
BORDA C yields 4

5.

```
∇ Z←COPELAND B;H
[1]   H←.5×1↓⍴B
[2]   Z←B+.<⍉B
[3]   Z←(H<Z)-(0<Z)∧Z<H
[4]   Z←+/Z
∇
```

8. **(a)**

```
∇ Z←MAJ B;I;N
[1]   N←1↓⍴B
[2]   Z←I×(N+1)≠I←((N÷2)<+/1=B)⍳1
∇
```

9.

```
∇ Z←REM2S N
[1]   Z←N÷2*¯1+(⍎((⌈2⊛N)⍴2)⊤N)⍳1
∇
```

12. Assuming index origin has the default value 1:

```
[3]   →(BEND,0,AEND)[2+×(F A)×F Z]
```

13. In index origin 1, all of (a)–(d) have the same effects. In index origin 0, (a)–(c) have the same effect, but (d) produces a branch to ⍳0 if *N* is not less than 10.

16.

```
∇ Z←THREE1TWO N
[1]   Z←,N
[2]   Z←Z,N←REMOVE2S 1+3×N
[3]   →2×(N≠1)∧N∧.≠¯1↓Z
∇
```

19. **(a)** *A* is *HERE IS SAM* and ⍴*A* is 11.
(b) *B* is *HERE 'IS SAM'* and ⍴*B* is 13.

22. Modify lines 6 and 7 of *ADAPINT* to get *ADAPINT3* and use *AIRS3*:

```
∇ ADAPINT3;A;B;INT
[1]   ⎕←'WARNING: THE INTEGRAND IS ASSUMED TO BE THE FUNCTION F'
[2]   ⎕←'PLEASE ENTER THE INTERVAL OF INTEGRATION AS A VECTOR:'
[3]   INT←⎕ ⋄ A←INT[1] ⋄ B←INT[2]
[4]   ⎕←'PLEASE ENTER THE DESIRED TOLERANCE:'
[5]   TOLERRPU←|⎕÷B-A
[6]   NRS←0 ⋄ INTAI←2⍴0 ⍝ INTAI IS LIST OF ACCEPTED INTERVALS
[7]   ZAI←A AIRS3 B          ⍝ ZAI CONTAINS ADAPTIVE APPROXIMATIONS
[8]   ⎕←'THE ESTIMATED INTEGRAL IS:'
[9]   ⎕←ZAI[1]
```

```
[10]  □←'THE NUMBER OF CALLS TO THE INTEGRATOR WAS:'
[11]  □←NRS
      ∇
```

AIRS3 is *AIRS* with changes on line 3, 6, and 7:

```
      ∇ Z←A AIRS3 B;M
[1]   ESTERR←|-/Z←A ROMSIMP B
[2]   NRS←NRS+1                          ⍝ COUNT CALLS TO ROMSIMP
[3]   →(ESTERR<TOLERRPU×|B-A)/ACCEPT
[4]   M←(A+B)÷2                          ⍝ IF NOT, FIND THE MIDPOINT
[5]   Z←(A AIRS M)+M AIRS B              ⍝ CALL AIRS ON HALF INTERVALS
[6]   →0
[7]  ACCEPT:INTAI←(A,B),[1]INTAI
      ∇
```

For (a) and tolerance $1E^{-}5$, the estimate is 3.141592661 using seven calls and accepting the four intervals

0 0.25
0.25 0.5
0.5 0.75
0.75 1

With tolerance $1E^{-}10$ the estimate is 3.141592654 with 141 calls and accepting 71 intervals.

23. **(a)**
```
1 2
3 4
```
(b)
```
*1 2*
*3 4*
```
(c) *N* = 1 2 3 4 5 *ETC*.

25.
```
A←(⍳15),⍉1.06 1.08 1.1∘.*⍳15
B←(17⍴5 1/1 0)⍀2 0 8 4 8 4 8 4⍪A
C←'YEAR 6%       8%       10%  ',[1]'-',[1],B
```

28. **(a)** 9 **(b)** *AAAAA* is the numeric matrix
```
1 2
3 4
```
(c)
```
3+3+3
    9
```

31. **(a)**
```
729 729 729
729 729 729
729 729 729

4782969 4782969 4782969
4782969 4782969 4782969
4782969 4782969 4782969

14348907 14348907 14348907
14348907 14348907 14348907
14348907 14348907 14348907
```

(b)
```
 729 ¯729  729
¯729  729 ¯729
 729 ¯729  729

 4782969 ¯4782969  4782969
¯4782969  4782969 ¯4782969
 4782969 ¯4782969  4782969
```

```
 14348907 -14348907  14348907
-14348907  14348907 -14348907
 14348907 -14348907  14348907
```

(c)
```
0.3449556 0.2948282 0.3602162
0.3435781 0.2953324 0.3610895
0.3441603 0.2951193 0.3607204

0.344262556  0.2950818717 0.3606555723
0.3442620376 0.2950820615 0.3606559009
0.3442622567 0.2950819813 0.360655762

0.3442623925 0.2950819316 0.360655676
0.344262199  0.2950820024 0.3606557986
0.3442622808 0.2950819725 0.3606557468
```

(d)
```
0.0895521 0.1479406 0.1239226
0.1137097 0.1878485 0.1573494
0.0773746 0.1278229 0.1070696

0.03003556673 0.04961881403 0.04156283173
0.03813773518 0.06300361191 0.05277450845
0.02595110586 0.04287127682 0.03591080721

0.02620176723 0.04328536988 0.03625766919
0.03326975878 0.05496170553 0.04603826518
0.02263865507 0.03739910174 0.03132708032
```

(e)
```
64  0 64
 0  1  0
64  0 64

16384     0 16384
    0     1     0
16384     0 16384

32768     0 32768
    0     1     0
32768     0 32768
```

(f)
```
 0.00390625  0.0234375  0.01953125
 0.00390625  0         -0.00390625
-0.00390625 -0.0078125 -0.00390625

 1.525878906E-5  9.155273438E-5  7.629394531E-5
 1.525878906E-5  0.000000000E0  -1.525878906E-5
-1.525878906E-5 -3.051757813E-5 -1.525878906E-5

 7.629394531E-6 -1.525878906E-5 -2.288818359E-5
 7.629394531E-6  3.051757813E-5  2.288818359E-5
-7.629394531E-6 -1.525878906E-5 -7.629394531E-6
```

32. (a)
```
5 4 3 2 1 0 -1
5 4 3 2 1 0 -1
1
```

(b)
```
5 6 7
  5 6 7
  1
```

Bibliography

APL News, Springer Verlag.

APL Quote Quad, Newsletter of SIGAPL, the Special Interest Group for APL, Association for Computing Machinery.

APL Quote Quad, The Early Years. Rockville, MD: APL Press, 1982.

Alvord, Linda, *Probability in APL*. Rockville, MD: APL Press, 1984.

Bergquist, Gary A., *APL Advanced Techniques and Utilities*. Vernon, CT: Zark, 1987.

Brown, James A., Sandra Pakin, and Raymond P. Polivka, *APL2 at a Glance*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Gilman, Leonard, and Allen J. Rose, *APL, An Interactive Approach,* Third edition. New York: Wiley, 1984.

Grenander, Ulf, *Mathematical Experiments on the Computer*. San Diego, CA: Academic Press, 1982.

Helzer, Garry, *Applied Linear Algebra with APL*. Boston: Little, Brown, 1983.

Iverson, Kenneth E., *A Programming Language*. New York: Wiley, 1962.

Iverson, Kenneth E., "Notation as a Tool of Thought", *Communications of the ACM*. August (1980).

Iverson, Kenneth E., "A Dictionary of APL", *APL Quote Quad, 18,* 5–40 (1987).

Orth, D. L., *Calculus in a New Key*. Rockville, MD: APL Press, 1976.

Peelle, Howard A., *APL, an Introduction*. New York: Holt, Rinehart & Winston, 1986.

Pommier, S., *An Introduction to APL*. New York: Cambridge University Press, English edition, 1983.

Ramsey, James B., and Gerald Musgrave, *APL-STAT: A Do-It-Yourself Guide to Computational Statistics Using APL*. New York: Lifetime Learning Publications, 1981.

Sims, Charles C., *Abstract Algebra, A Computational Approach*. New York: Wiley, 1984.

Turner, Jerry R., *APL is EASY!* New York: Wiley, 1987.

References

Burden, Richard L., and J. Douglas Faires, *Numerical Analysis,* Fourth Edition. Boston: PWS-Kent, 1989.

Feller, William, *An Introduction to Probability Theory and its Applications*. New York: Wiley, 1950.

Lewis, P. A. W., and E. J. Orav, *Simulation Methodology for Statisticians, Operations Analysts, and Engineers,* Volume 1. Pacific Grove, CA: Brooks/Cole, 1989.

Magyar, Zoltan, "A Recursion on Quadruples", *American Math Monthly, 91,* 360–362 (1984).

Pleuss, Vera, *Introduction to the Theory of Error-Correcting Codes*. New York: Wiley-Interscience, 1982.

Straffin, Philip, *Topics in the Theory of Voting*. Cambridge, MA: Birkhäuser, 1980.

Tucker, Alan, *A Unified Introduction to Linear Algebra, Models, Methods and Theory*. New York: Macmillan, 1988.

Warshall, S., "A Theorem on Boolean Matrices", *Journal of the ACM,* 11–12 (1962).

Index